NPS-TE-11-181

ACQUISITION RESEARCH
SPONSORED REPORT SERIES

Mathematical Modeling to Reduce the Cost of Complex System Testing: Characterizing Test Coverage to Assess and Improve Information Return

21 September 2011

by

Dr. Karl D. Pfeiffer, Assistant Professor,

Dr. Valery A. Kanevsky, Research Professor, and

Dr. Thomas J. Housel, Professor

Graduate School of Operational & Information Sciences

Naval Postgraduate School

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **21 SEP 2011**	2. REPORT TYPE	3. DATES COVERED **00-00-2011 to 00-00-2011**

4. TITLE AND SUBTITLE **Mathematical Modeling to Reduce the Cost of Complex System Testing: Characterizing Test Coverage to Assess and Improve Information Return**	5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Graduate School of Operational and Information Sciences,Monterey,CA,93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S) 11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Effective, cost-efficient testing is critical to the long-term success of Open Architecture within the Navy?s Integrated Warfare System. In previous research we have developed a simple, effective framework to examine the testing of complex systems. This model and its prototype decision aid provide a rigorous yet tractable approach to improve system testing, and to better understand and document the system and component interdependencies across the enterprise. An integral part of this model is characterizing test coverages on modules. Using idealized simulations of complex systems, we investigate the sensitivity of test selection strategy to the precision with which these coverages are specified. Monte Carlo analysis indicates that best-test selection strategies are somewhat sensitive to the precision of test coverage specification, suggesting significant impact on testing under fixed-cost constraint. These results provide significant insight as we extend this work with further study of real-world systems by applying, and refining, the mathematical analysis and computer simulation within this framework. The current decision-aid software will be further developed using these operational test and evaluation data improving the fidelity of the current modeling while making available to program managers and system designers a usable and relevant tool for test?retest decisions.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**87**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

The research presented in this report was supported by the Acquisition Chair of the Graduate School of Business & Public Policy at the Naval Postgraduate School.

Abstract

Effective, cost-efficient testing is critical to the long-term success of Open Architecture within the Navy's Integrated Warfare System. In previous research we have developed a simple, effective framework to examine the testing of complex systems. This model and its prototype decision aid provide a rigorous yet tractable approach to improve system testing, and to better understand and document the system and component interdependencies across the enterprise. An integral part of this model is characterizing test coverages on modules. Using idealized simulations of complex systems, we investigate the sensitivity of test selection strategy to the precision with which these coverages are specified. Monte Carlo analysis indicates that best-test selection strategies are somewhat sensitive to the precision of test coverage specification, suggesting significant impact on testing under fixed-cost constraint. These results provide significant insight as we extend this work with further study of real-world systems by applying, and refining, the mathematical analysis and computer simulation within this framework. The current decision-aid software will be further developed using these operational test and evaluation data, improving the fidelity of the current modeling while making available to program managers and system designers a usable and relevant tool for test–retest decisions.

Keywords: Open Architecture, Navy's Integrated Warfare System, testing of complex systems, Monte Carlo analysis

THIS PAGE INTENTIONALLY LEFT BLANK

About the Authors

Karl D. Pfeiffer is a Visiting Assistant Professor of Information Sciences at the Naval Postgraduate School. His current research interests include decision-making under uncertainty, particularly with regard to command and control (C2) systems; stochastic modeling of environmental impacts to weapons and communication systems; and probability modeling and numerical simulation in support of search, identification and pattern recognition applications (e.g., complex system testing, allocation of effort for reconnaissance).

Karl D. Pfeiffer
Graduate School of Operational and Information Sciences
Naval Postgraduate School
Monterey, CA 93943-5000
Tel: 831-656-3635
Fax: (831) 656-3649
E-mail: kdpfeiff@nps.edu

Valery A. Kanevsky is a Research Professor of Information Sciences at the Naval Postgraduate School. His research interests include probabilistic pattern recognition; inference from randomly distributed inaccurate measurements, with application to mobile communication; patterns and image recognition in biometrics; computational biology algorithms for microarray data analysis; and Kolmogorov complexity, with application to value allocation for processes without saleable output. Another area of interest is in the so-called needle-in-a-haystack problem: searching for multiple dependencies in activities within public communication networks as predictors of external events of significance (e.g., terrorist activities).

Thomas J. Housel is a Professor of Information Sciences at the Naval Postgraduate School. Professor Housel specializes in valuing intellectual capital, knowledge management, telecommunications, information technology, value-based business process re-engineering, and knowledge value measurement in profit and non-profit organizations. His current research focuses on the use of knowledge-value added (KVA) and real options models in identifying, valuing, maintaining, and exercising options in military decision-making. His work on measuring the value of intellectual capital has been featured in a *Fortune* cover story (October 3, 1994) and *Investor's Business Daily*, numerous books, professional periodicals, and academic journals (most recently in the *Journal of Intellectual Capital*, 2005).

Thomas J. Housel
Department of Information Science
Graduate School of Operational and Information Sciences
Monterey, CA 93943
Phone: 831-656-7657
Email: tjhousel@nps.edu
Web: faculty.nps.edu/tjhousel

NPS-TE-11-181

ACQUISITION RESEARCH
SPONSORED REPORT SERIES

Mathematical Modeling to Reduce the Cost of Complex System Testing: Characterizing Test Coverage to Assess and Improve Information Return

21 September 2011

by

Dr. Karl D. Pfeiffer, Assistant Professor,

Dr. Valery A. Kanevsky, Research Professor, and

Dr. Thomas J. Housel, Professor

Graduate School of Operational & Information Sciences

Naval Postgraduate School

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

I. Overview

In previous research we have developed a framework for describing the performance of a test suite for assessment of a complex system under repair or under routine maintenance (Pfeiffer, Kanevsky, & Housel, 2009a, 2009b, 2010). This model was then implemented in a decision support tool to investigate strategies for test selection under fixed cost or fixed reliability constraints. Construction of the model for simulation required the characterization of test coverages on modules; that is, we needed an a priori estimate of how much of the module was exercised by a particular test.

For hardware modules, this test coverage is reasonably simple to estimate (see, for example, Barford, Kanevsky & Kamas, 2004). For software systems, however, the notion of test coverage is more problematic and may require more knowledge of the internal structure of the modules and their interdependencies (Zhu, Hall, & May, 1997). Although the notion of software testing is well studied, the characterization of test coverage can vary widely among investigators (compare, for example, Leung & White, 1991; White & Leung, 1992; Weyuker, 1998; Tsai, 2001; Rothermel, Untch, & Harrold, 2001; Mao & Lu, 2005). Often, internal knowledge of hardware and software modules is not available to developers of integrated test suites, particularly with commercial off-the-shelf (COTS) technologies. The increasing use of COTS in current weapons systems (Caruso, 1995; Dalcher, 2000), coupled with the complexity of end-to-end systems (Athans, 1987; Brazet, 1993), suggests that characterizing test coverages in an open architecture system will remain a significant challenge.

How important are these test coverages to developing an effective test strategy? That is, how precisely and how accurately must we specify these coverages to evaluate effective test strategies? Extending our previous work, we investigate the sensitivity of test selection strategy to the characterization of test coverages within the system under test. Using an analytic approach to inform further modeling and simulation work, we seek to better understand how well we must specify these a priori coverage estimates in order to derive useful testing strategies.

The rest of this paper is organized as follows: Section 2 briefly reviews the framework we have developed for investigating testing strategies; Section 3 discusses the analytic background and simulation approach in examining the sensitivity of information returned to the coverage specification; Section 4 describes simulation results and significant findings; and Section 5 discusses future avenues for research.

II. Background

In the present discussion, we define testing as the mechanism by which we trade some fixed cost (e.g., time, money) for information about the state of subcomponents and overall reliability of our system (Figure 1). In general, we seek the maximum information available for the minimum cost.

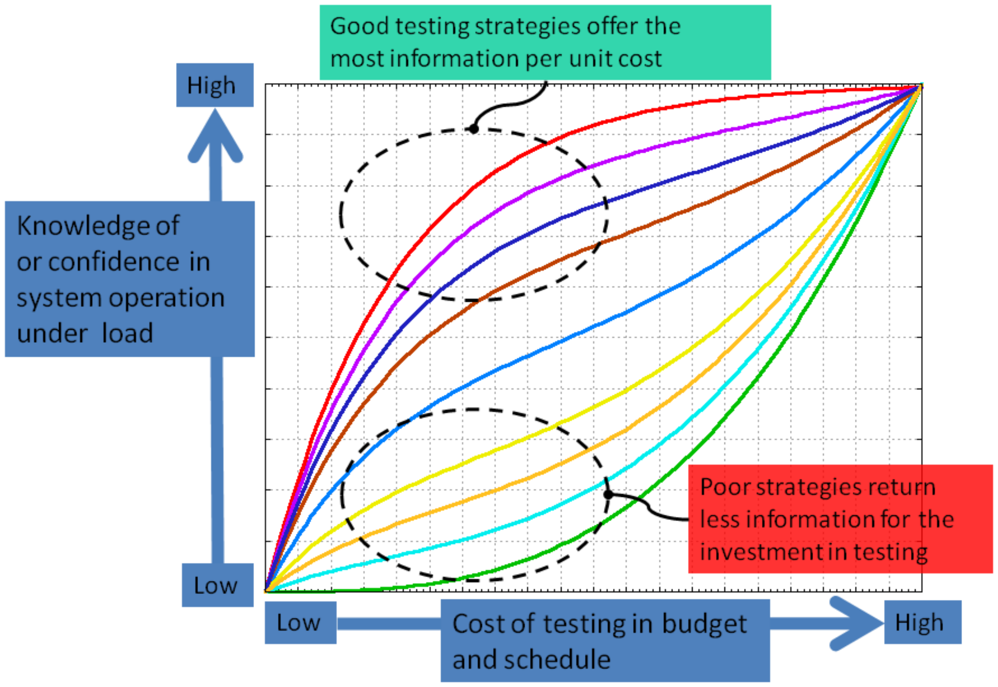

Figure 1. An Idealized Representation of Testing Strategies in Terms of Information Returned for Testing Accomplished
Note. Each solid line represents a particular testing strategy, with better strategies distinguished by steeper ascent or greater information return per unit cost.

Mathematical models proposed by von Neumann (1952) and Moore and Shannon (1956a; 1956b) shaped much of the early work on component and system reliability. An early focus on fault diagnosis, particularly in electro-mechanical systems, characterized work by Sobel and Groll (1966), Butterworth (1972), Garey (1972), Fishman (1990), and others, in what is often known as the test-sequencing problem. That is, which test sequence most cost-efficiently arrives at a correct diagnosis in a failed system?

In software engineering, we are most often faced not with a failed system, but with a large system undergoing maintenance. Testing in this situation is used to establish that no defect has been added to the system by these engineering upgrades. This regression testing, or test–retest dilemma, can be more difficult than diagnostic testing of a failed system because by its nature testing cannot absolutely demonstrate that no defect exists (Dijkstra, 1972). A good test suite and good testing strategy, however, can often demonstrate that a defect is *highly unlikely* in the system under test (Zhu, Hall & May,1997).

In previous work (Pfeiffer et al., 2009a, 2009b, 2010) we have developed a unified modeling framework with risk and cost as the common tension regulating the degree of testing required.   The cost of testing can be evaluated in terms of dollars, or time, or both, with an assumption that more testing is generally more costly; it is not true in general, however, that more testing always increases our knowledge of the state of our system. This knowledge is tied to risk. In this context, risk refers to the degree of certainty we can achieve (or ambiguity we can eliminate) within a fixed cost constraint or within the power or sensitivity of a given test suite.

We characterize our system under test **S** as a collection of modules $\{M_i\}$, and a suite of tests $\{T_x\}$ used to interrogate these modules (Figure 2). These tests are our means to identify defective modules, or, in the case of test–retest, to establish with high probability that no defects exist. We assume that tests return ambiguous information about the state of modules within the system; that is, no single test is likely to return perfect knowledge about a particular module.

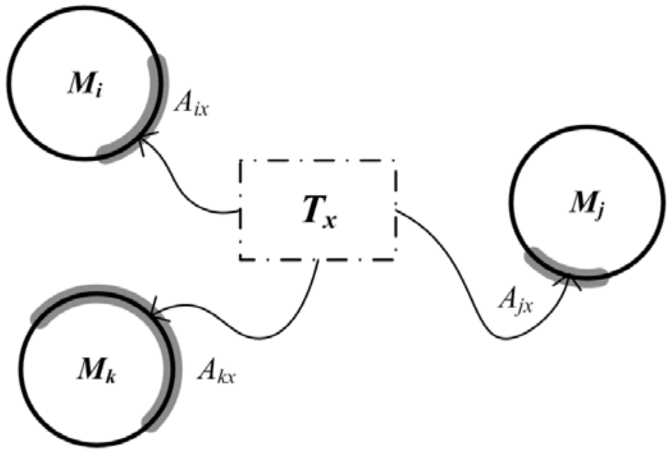

Figure 2. Notional Depiction of the Coverage of T_x on S, With Multiple Modules Exercised by This Test

Note. A FAIL result from T_x indicates that at least one of the subset $\{M_i, M_j, M_k\}$ has failed.

Each test is assumed to exercise several modules (Figure 2), and several tests may exercise the same module. In the case of several tests covering a particular module, the framework easily accounts for overlapping and disjoint coverages (Figure 3).

The model framework described in Pfeiffer et al. (2009a, 2009b, 2010) presents a useful and realistic ambiguity in two aspects. The first is that a test is rarely assumed to cover or exercise all functionality of a module. This means that when a particular test T_x passes, we know only that the region exercised by the test does not contain a defect; a defect may still exist in those regions not inspected by the test (Figure 2). A second ambiguous aspect is that when test T_x fails, several modules may be at fault (Figure 2), though such a result should significantly reduce the number of suspect modules in **S**.

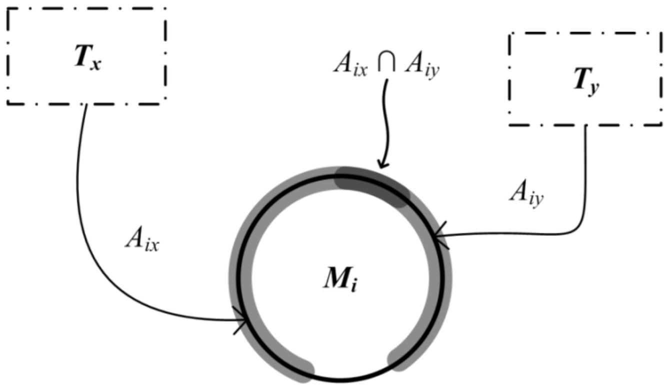

Figure 3. Overlapping coverage between tests T_x and T_y

The vector arcs specifying test coverage are intended to lend precision to the model specification and implementation. With these vector artifacts, the overlap among tests on a single module can be precisely specified, and the disjoint regions can be similarly specified (Figure 3). In the original work (Pfeiffer et al., 2009a), we proposed that subject-matter experts could estimate these coverage data as a starting point for further modeling and simulation work. In the present study, we further examine how precise these estimates should be to deliver meaningful decision support for cost-effective test strategies.

III. Analytic Modeling and Computer Simulation Approach

We characterize our knowledge of the system under test **S** as a vector of probabilities {b_i} that any given module M_i is bad. Our knowledge of the system is perfect when every b_i is either 0 (absolutely good) or 1 (absolutely bad). In practice, we are unlikely to see perfect results (e.g., b_i = 0 or 1) though we can, with a well-designed test suite and an effective test strategy, minimize the residual information entropy of the vector (Pfeiffer et al., 2009a). This entropy is defined following Shannon (1948):

$$h_i = -b_i \log_2 b_i - (1-b_i)\log_2(1-b_i) \tag{1}$$

The initial values for {b_i} are assumed to be available from subject-matter experts or a priori failure rate estimates. Our simulation work has demonstrated that test strategy outcomes are relatively insensitive to these initial {b_i} because of the iterative nature of this approach. That is, after a few tests have been executed, the initial vector {b_i} moves significantly towards lower entropy (Pfeiffer et al., 2010). The test coverages connecting the tests {T_x} to the modules {M_i} appear to be the more relevant initial criteria in these simulations of system testing. This is another motivation for the present study.

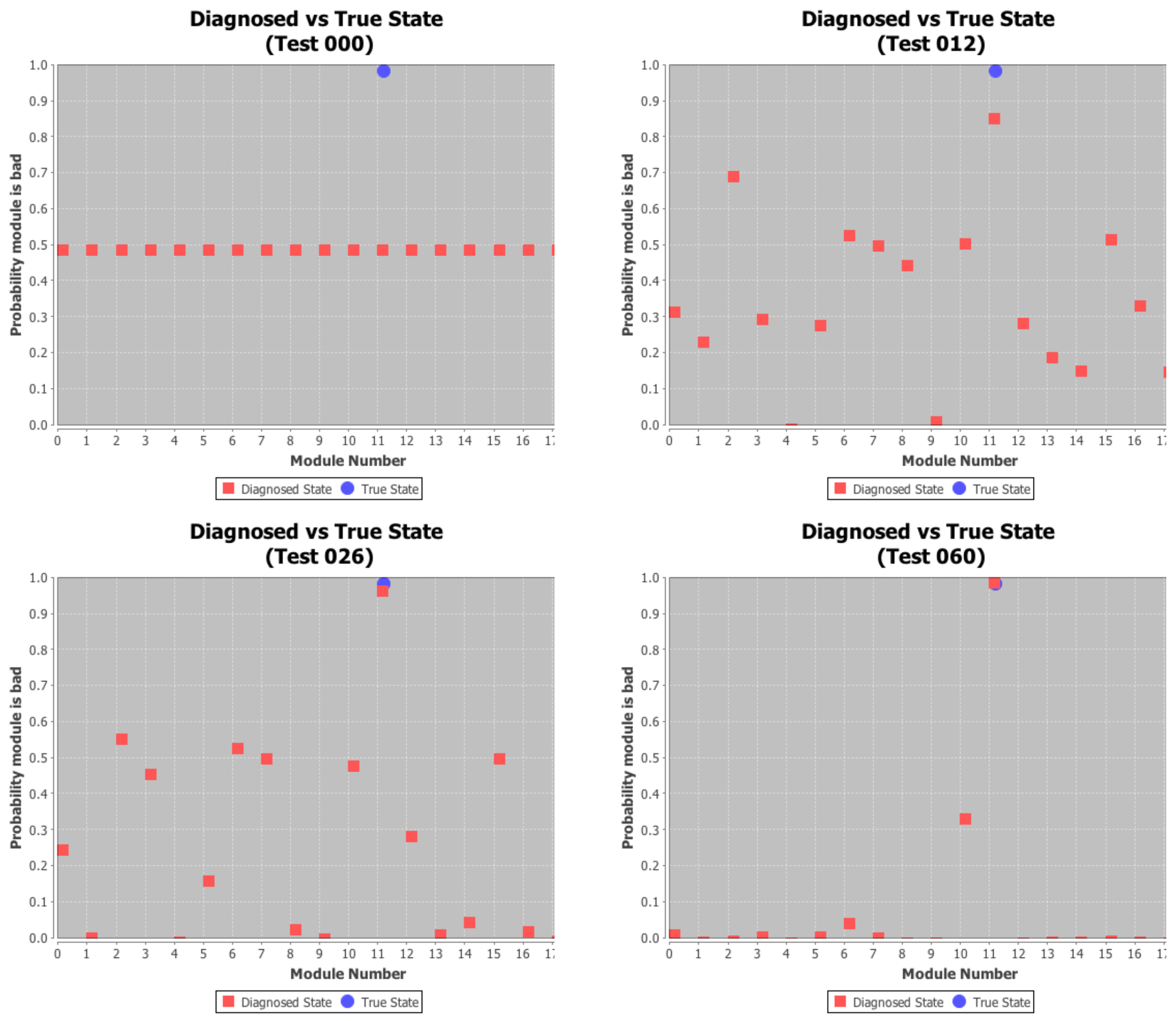

Figure 4. Diagnostic sequence or trial from the Monte Carlo simulation of testing.

The decision support tool developed in Pfeiffer et al. (2009a) simulates the testing of a complex system using minimal descriptions of tests, modules, and their connecting coverages. For idealized simulations, a range of coverages is specified between tests and modules, coupled with a target number of tests per module and modules per test. Simulations may be run with zero or more defects. The zero-defect case is particularly important for investigating test–retest or regression cases.

Each simulation typically involves a large number of trials (notionally 100 to 1000) and this facilitates examining the bounds of the idealized assumptions. The diagnostic sequence from a single trial is depicted in Figure 4. The reduction in residual entropy across the system is apparent as testing progresses from the initial state (Figure 4, upper left) to a useable diagnosis (Figure 4, lower right) with the defective module correctly identified. The system entropy is computed as the aggregate of residual entropy associated with each module:

$$H = \sum_i -b_i \log_2 b_i - (1-b_i)\log_2(1-b_i) \qquad (2)$$

After execution of a test, we update the prior probability *b_i* of each module M_i to the new probability b_i' based on the test outcome (PASS or FAIL):

$$b_i' = \begin{cases} \mathrm{P}(B_i \mid P_x) & \text{if } T_x \text{ passes} \\ \mathrm{P}(B_i \mid F_x) & \text{if } T_x \text{ fails} \end{cases} \qquad (3)$$

These conditional probabilities are connected to test coverages through the Bayesian relations:

$$\mathrm{P}(B_i \mid P_x) = \frac{\mathrm{P}(P_x \mid B_i)\mathrm{P}(B_i)}{\mathrm{P}(P_x)} = \left(\frac{\mathrm{P}(P_x \mid B_i)}{\mathrm{P}(P_x)}\right) b_i \qquad (4)$$

$$\mathrm{P}(B_i \mid F_x) = \frac{\mathrm{P}(F_x \mid B_i)\mathrm{P}(B_i)}{\mathrm{P}(F_x)} = \left(\frac{\mathrm{P}(F_x \mid B_i)}{\mathrm{P}(F_x)}\right) b_i \qquad (5)$$

And these probabilities are computed with:

$$\mathrm{P}(P_x) = \prod_{i=1}^{n} \left[1 - \alpha_{ix} b_i\right] \qquad (6)$$

$$\mathrm{P}(F_x) = 1 - \prod_{i=1}^{n} \left[1 - \alpha_{ix} b_i\right] \qquad (7)$$

Knowledge of the coverage dyad $\{\alpha_{ix}\}$ is thus intrinsic to minimizing system entropy (Equation 2) and developing a cost-effective strategy for system testing. How precisely must these coverages be specified to be useful, though?

IV. Simulation Results

In previous work (Pfeiffer et al., 2009a, 2009b) we have investigated the relative performance improvement in testing strategies using a best next test (one-test look ahead) and best next two tests (two-test look ahead). The coverages for these investigations were constructed by sampling a uniform distribution on [0.1,0.9] for each connected test and module pair.

Results from Pfeiffer et al. (2010) suggest that the best-next-two-tests strategy offers some improvement over a one-test look ahead; though the time required developing the two-test look ahead is on the order of 2.5 times the one-test strategy. A random test selection strategy was also used in this work as a baseline or no-strategy approach. Both best and best-two strategies clearly outperformed this random approach.

In this work, we also introduced an equivalent metric to residual information entropy (Equation 1) using instead the maximum probability q_i related to b_i by:

$$q_i = \max(b_i, 1 - b_i) \tag{8}$$

This measure is more intuitive than Equation 2 and represents an expected value of a replacement (or maintenance) decision with respect to a particular module. If, for example, a particular module has a $b_i = 0.70$, we may replace it knowing that this informed guess should be correct 70% of the time. This also means that in 30% of these cases we will unnecessarily replace or perform more granular debugging on this module. Our number of correct diagnoses across the system will increase as each b_i is adjusted, by testing, away from $b_i = 0.5$ towards either 0 or 1 (Figure 4). In Pfeiffer et al. (2009a), we have shown that minimizing system entropy is approximately equivalent to maximizing the number of correct diagnoses.

In evaluating the best next test (or best next two tests), the measure (Equation 8) is aggregated as a system measure for a particular test T_x:

$$Q(T_x) = \sum_{i=1}^{n} q_{ix} \tag{9}$$

At any point in diagnostic testing, all available T_x are evaluated with Equation 9, and the largest $Q(T_x)$ indicates the next best T_x. The conditional probabilities dependent upon the specification of coverage (Equations 3–7) are intrinsic to this computation.

In simulation work using the decision support tool for complex testing, we examined the sensitivity of test strategies to the specification of test coverage within the model. Specifically, we examined both random and best-next test strategies with different specifications of coverage about a mean coverage per module of 0.7 or 70%. All $\{b_i\}$ were initialized with a maximum entropy value of $b_i = 0.5$, consistent with our assertion that the iterative simulation is relatively insensitive to the initial $\{b_i\}$ (Pfeiffer et al., 2009a). All runs were made with zero defects present, to emphasize the utility of this work for test–retest or regression scenarios.

The fundamental connection of coverage to information (Equations 2, 6, and 7) suggests that, in general, more coverage per test should yield more information. For this investigation, four specifications were used: a uniformly distributed coverage among tests and modules from 50% to 90%, or [0.5,0.9]; and, fixed coverages of 50%, 70%, and 90%. A nominal 300 trials were used for this work, though the model output statistics were examined for 1,000 trials without significant difference.

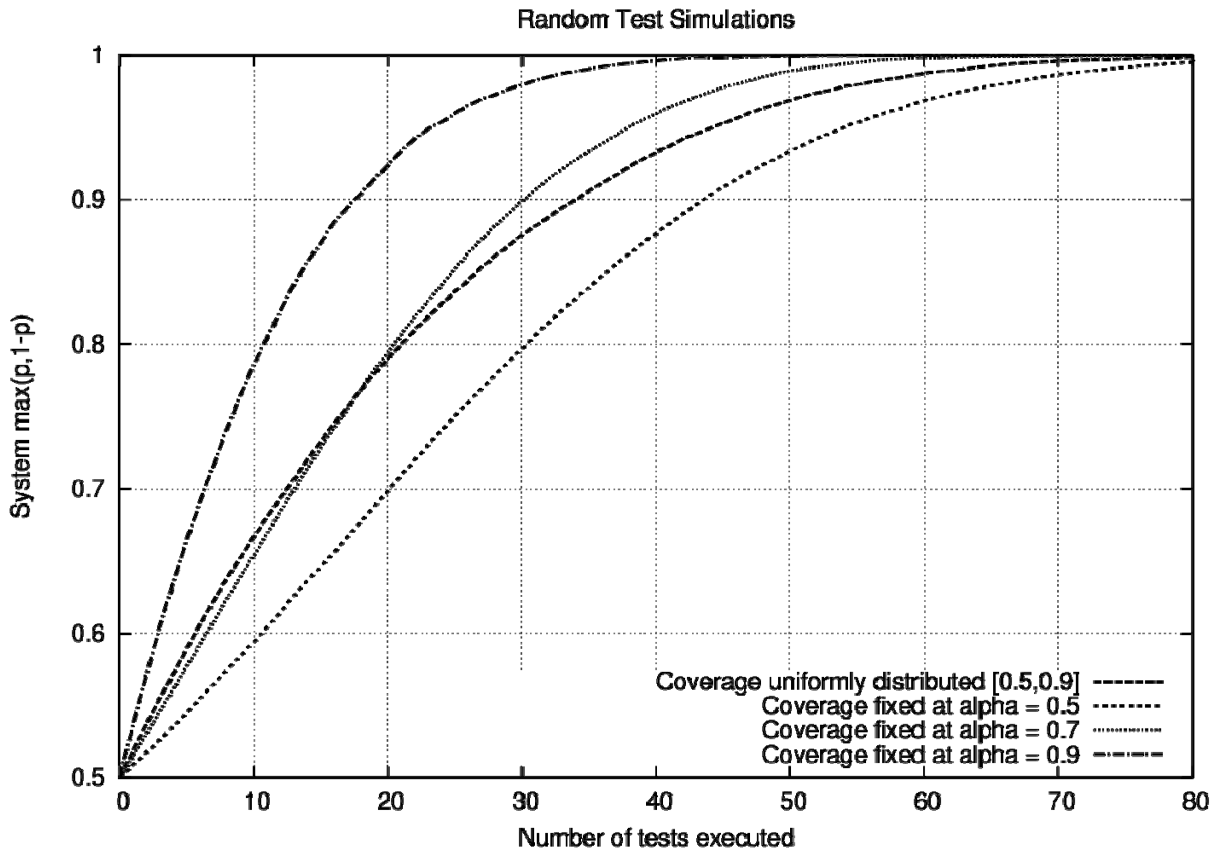

Figure 5. Simulation results using different coverage specifications

The random strategy simulations (Figure 5) do indeed show more information returned when the coverage is fixed at 90%, and significantly less information returned when the coverage is fixed at 50%. Perhaps more interesting is the comparison of fixed coverage at 70% with a random coverage on the interval [0.5, 0.9], which has a mean of 70%. These runs for the random strategy appear quite similar up to about the first 20 tests (Figure 5). At this point, the fixed coverage at 70% appears to outperform the random coverage on [0.5, 0.9].

In contrast to the no-strategy approach, the best next test simulations (Figure 6) show pronounced differences among coverage specifications. The fixed 90% coverage run appears somewhat better in information returned per test execution, though interestingly the 70% and 50% coverage runs appear to underperform compared to the random simulation (Figure 5). These differences are not consistent over the test execution profile, however. This is particularly interesting because both the random and best next simulations were run with random number generators seeded identically; thus, the differences highlighted between Figures 5 and 6 are solely a function of the differences in the rate of information returned by the two strategies.

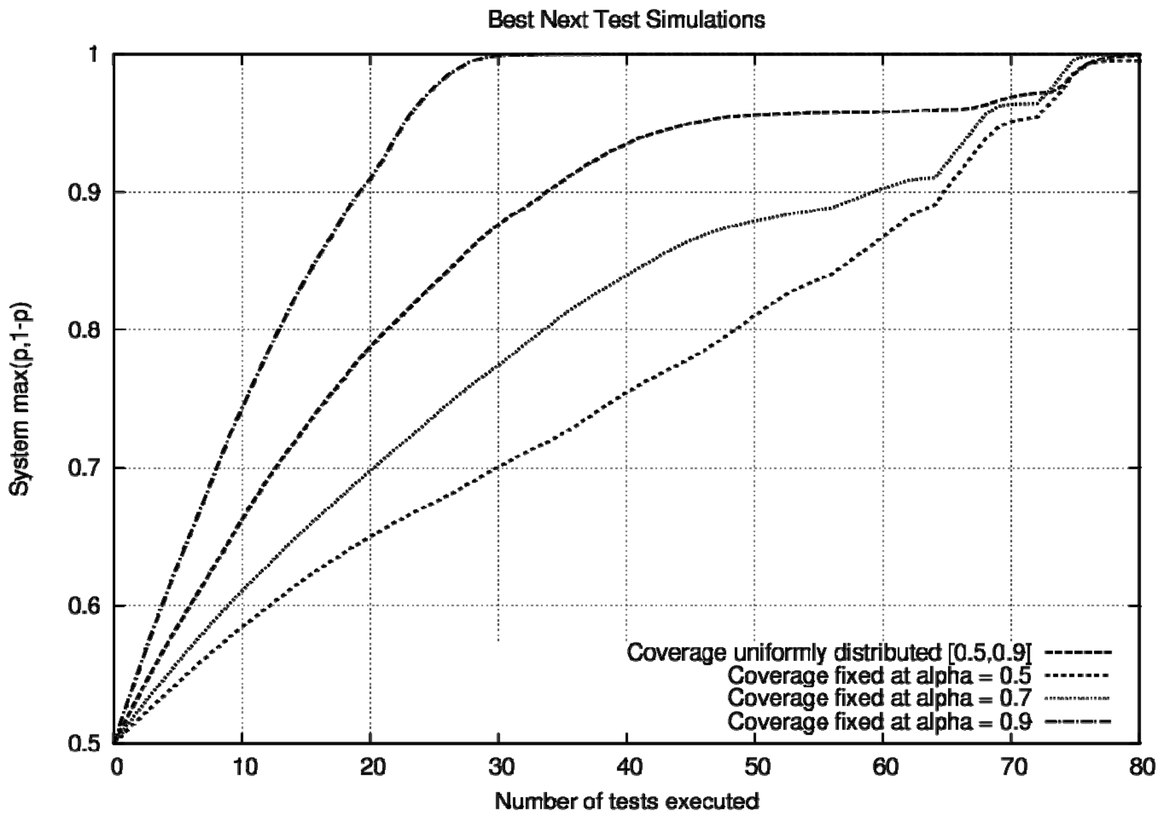

Figure 6. Simulation results using different coverage specifications

All of these simulations were conducted with no defects present, and identification of a defect tends to sharply alter the information profile in a run; intuitively, this is because the first FAIL result in test execution should sharply reduce the number of suspect modules across the system. In the absence of defects, it is possible, particularly as the testing progresses and alters the vector $\{b_i\}$ that the differences among tests in information returned (Equation 9) may vary widely on a one-test look ahead.

Consistent with our previous studies (e.g., Pfeiffer et al., 2010), we made a two-test look-ahead simulation to better assess the sensitivity of coverage specification to test selection strategy. Overall results (Figure 7) show little improvement from the one-test strategy (Figure 6), though the best performer (fixed coverage at 90%) does show some early improvement over the first ten tests executed. These results do suggest that the effectiveness of a test selection strategy is connected to the precision with which the test coverages are specified.

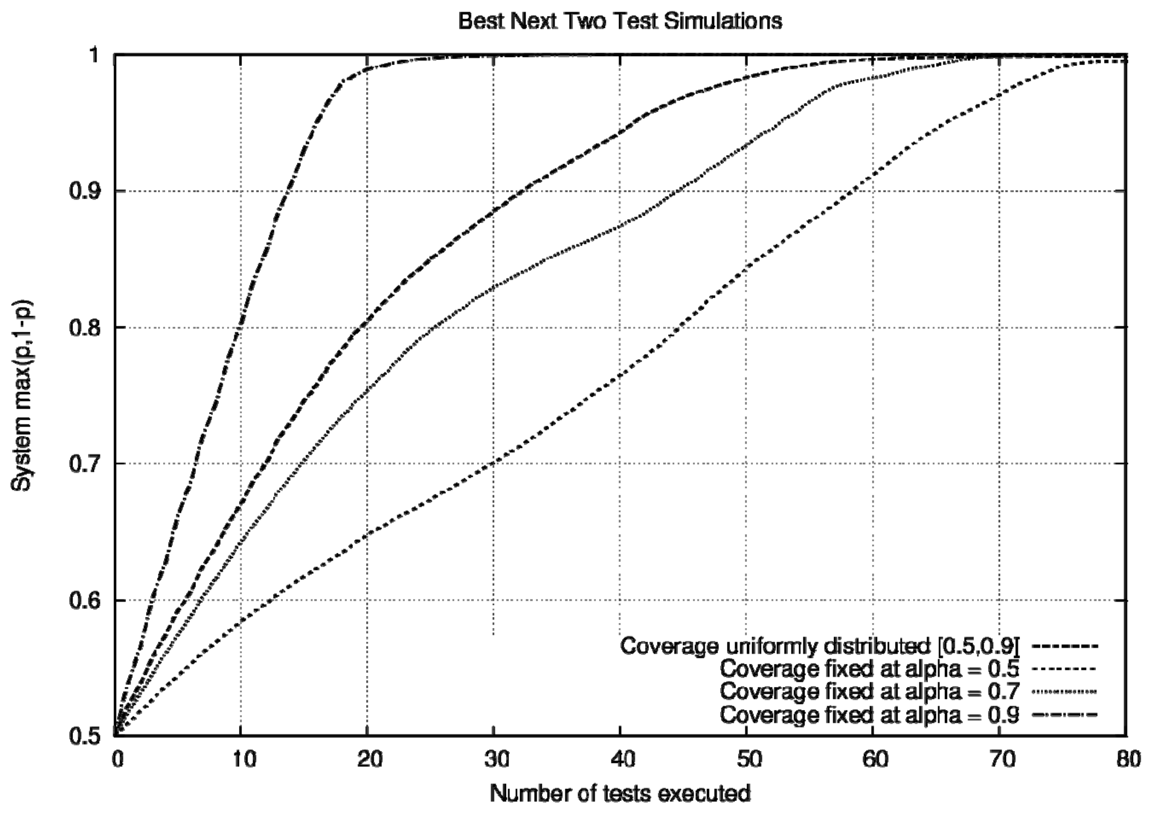

Figure 7. Simulation results using different coverage specifications for a best next two-test selection.

We should keep in mind that these idealized simulations place no constraint on the overlap among coverages on tests. We verified in simulation log files that significant overlap among tests (e.g., Figure 3) increased as the fixed coverage progressed from 50% to 90%. The impact of this overlap on test selection appears most dramatic in the non-random simulations (Figures 6 and 7) as best next and best-next-two strategies make better use of the information returned by each test.

This overlap also means that many or most of the modules in the idealized system were completely covered by some number of tests in the test suite, leading to the near perfect information after about 30 tests have been executed (Figures 6 and 7). We expect real-world systems would rarely achieve perfect coverage regardless of the number of tests available because of the nature of complex systems. For anything but a trivial component or module, we are unlikely to construct a set of tests that cover *all* branch paths or all of the input and output space.

An obvious conclusion from these results is that more coverage per test appears to improve the testing process. While this result may be somewhat intuitive, an equally useful and interesting result is that better specification of coverages increases the benefits from a rigorous test selection strategy. Further investigation with both real coverage data from operational systems and idealized simulations with more complex distributions of coverage should yield additional insights into this problem.

V. Conclusions and Future Work

Effective, cost-efficient testing and re-testing is critical to the long-term success of Open Architecture. Using the framework for complex system testing developed in Pfeiffer et al. (2009b), we have conducted additional simulation work to examine the sensitivity of test selection strategies to the specification of test coverages. Characterization of test coverages, particularly for software-intensive systems, remains a difficult challenge (Zhu, Hall & May, 1997), though in this work we did not address this problem directly. Rather, in the framework of our existing model, we have examined the impact of precision in specifying coverage on the information returned per test.

Not surprisingly, the test selection strategies we have investigated are quite sensitive to various specifications of test coverage. In these idealized simulations, less precision in coverage specification appears to flatten the information returned per test. Incorporation of more real test data from operational systems should help with further investigation of this point. The idealized work permitted complete (100%) or near-complete coverage of modules with overlapping tests (particularly for fixed coverages of 70% to 90%) that would be unlikely in real-world testing. In fact, we speculate that in simulating real-world systems, we will likely encounter test scenarios in which no module is completely covered by testing, and real coverages are at best 95% to 98% with all overlapping coverages considered.

The decision support tool used in these simulations could also be further refined to permit specifying test-to-module coverages in terms of a collection of triangle or uniform distributions. This should better capture subject-matter expertise in a quantitative manner. For example, a quasi-idealized simulation of a Garage Door Opening System could work with a specification that the *Object Detection Test* exercised at least 30% of the *Remote Control Module*, though no more than 50%, with a mode or mean of 40%. While these numbers may still be speculative or notional on the part of the subject-matter expert, these confidence bounds would represent useful input to the overall test selection strategy.

References

Athans, M. (1987). Command and control (C2) theory: A challenge to control science. *IEEE Transactions on Automatic Control, 32*(4), 286–293.

Barford, L., Kanevsky, V., & Kamas, L. (2004). Bayesian fault diagnosis in large-scale measurement systems. In *Proceedings of the IMTC 2004: Instrumentation and Measurement Technology Conference* (pp. 1234–1239). Como, Italy: IEEE.

Brazet, M. D. (1993). AEGIS ORTS—The first and future ultimate integrated diagnostics system. *Aerospace and Electronic Systems Magazine, 9*(2), 40–45.

Butterworth, R. (1972). Some reliability fault-testing models. *Operations Research, 20*(2), 335–343.

Caruso, J. (1995). The challenge of the increased use of COTS: A developer's perspective. In *Proceedings of the Third Workshop on Parallel and Distributed Real-Time Systems* (pp. 155–159). Santa Barbara, CA: IEEE.

Dalcher, D. (2000). Smooth seas—Rough sailing: The case of the lame ship. In *Proceedings of the Seventh International Conference on Engineering of Computer Based Systems (EBCS 2000*; pp. 393–395). Edinburgh, Scotland: IEEE.

Dijkstra, E. (1972). Notes on structured programming. In O.-J. Dahl, E. Dijkstra, & C. Hoare (Eds.), *Structured programming* (pp. 1–72). London, UK: Academic Press.

Fishman, G. S. (1990). How errors in component reliability affect system reliability. *Operations Research, 38*(4), 728–732.

Garey, M. (1972). Optimal binary identification procedures. *SIAM Journal on Applied Mathematics, 23*(2), 173–186.

Leung, H., & White, L. (1991). A cost model to compare regression test strategies. In *Proceediings of the Conference on Software Maintenance* (pp. 201–208). Sorrento, Italy: IEEE.

Mao, C., & Lu, Y. (2005). Regression testing for component-based software systems by enhancing change information. In *Proceedings of the 12th Asia–Pacific Software Engineering Conference (APSEC'05*; pp. 1–8). Taipei, Taiwan: IEEE.

Moore, E., & Shannon, C. (1956a). Reliable circuits using less reliable relays, Part I. *Journal of the Franklin Institute*, 191–208.

Moore, E., & Shannon, C. (1956b). Reliable circuits using less reliable relays, Part II. *Journal of the Franklin Institute*, 281–298.

Pfeiffer, K. D., Kanevsky, V. A., & Housel, T. J. (2009a). *Reducing the cost of risk-based testing: Management of testing options to manage risk in test and evaluation.* Monterey, CA: Naval Postgraduate School, Acquisition Research Program.

Pfeiffer, K. D., Kanevsky, V. A., & Housel, T. J. (2009b). Testing of complex systems. In *Proceedings of the INFORMS Annual Meeting* (pp. 130–135). San Diego, CA: INFORMS.

Pfeiffer, K. D., Kanevsky, V. A., & Housel, T. J. (2010). An information–theoretic approach to software test–retest problems. In *Proceedings of the Seventh Annual Acquisition Research Symposium* (pp. 100–120). Monterey, CA: Naval Postgraduate School.

Rothermel, G., Untch, R. H., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, 27(10), 929–948.

Shannon, C. (1948). A mathematical theory of communication. Bell System Technical Journal, 27, 379–423; 623–656.

Sobel, M., & Groll, P. (1966). Binomial group-testing with an unknown proportion of defectives. Technometrics, 8(4), 631–656.

Tsai, W. (2001). End-to-end integration testing design. 25th Annual International Computer Software and Applications Conference (COMPSAC) (pp. 166-171). Chicago, IL: IEEE.

von Neumann, J. (1952). Probabilistic logics and synthesis of reliable organisms from unreliable components. In Automata Studies (AM-34) (pp. 45-98). Princeton, NJ: Princeton University Press.

Weyuker, E. (1998). Testing component-based software: A cautionary tale. IEEE Software, 15(5), 54–59.

White, L., & Leung, H. (1992). A firewall concept for both control-flow and data-flow in regression integration testing. In Proceedings of the Conference on Software Maintenance (pp. 262–270). IEEE., 9-12 Nov, Orlando, FL.

Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM Computing Surveys, 29(4), 366–427.

Appendix A. Java Source Code Tree for the Risk-Based Testing Simulation (rbts.jar)

The source code for the Risk-based Testing Simulation (rbts.jar) is archived as a Java NetBeans Project and is available upon request from Tom Housel (tjhousel@nps.edu) or Karl Pfeiffer (karl.pfeiffer@yahoo.com). The source code tree is arranged as follows:

```
src/rbts
src/rbts/Main.java
src/rbts/resources
src/rbts/resources/inherit.gif
src/rbts/package-list

src/rbts/common
src/rbts/common/BitMatrix.java
src/rbts/common/Environment.java
src/rbts/common/Logger.java
src/rbts/common/TrianglePDF.java
src/rbts/common/Utility.java

src/rbts/obj
src/rbts/obj/Configuration.java
src/rbts/obj/Coverage.java
src/rbts/obj/Module.java
src/rbts/obj/Probe.java
src/rbts/obj/SystemObject.java
src/rbts/obj/Test.java

src/rbts/ui
src/rbts/ui/AnimationPanel.java
src/rbts/ui/ConfigurationPanel.java
src/rbts/ui/Overseer.java
src/rbts/ui/SimulationAnalysisPanel.java
src/rbts/ui/SystemObjectPanel.java
```

```
src/rbts/rbts
src/rbts/rbts/Main.html
src/rbts/rbts/package-frame.html
src/rbts/rbts/package-summary.html
src/rbts/rbts/package-tree.html
```

In subsequent appendices we will show selected source code, though the complete text is available upon request either in hard or soft copy.

Appendix B. Java Source Code for Simulation Objects (src/rbts/obj)

B.1 Configuration.java

```
//
//  =====================================================================
//
//  Module: Configuration.java
//
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2011
//  =====================================================================
//
package rbts.obj;

import rbts.common.*;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.xml.sax.SAXException;

import java.io.*;
import java.util.ArrayList;
import java.util.Random;

//
//  package: Configuration.java
//  ---------------------------
//
//    This is the smallest unit of simulation execution
//
public class Configuration {

    //
    //  ... attributes from file ...
    //
    String CaseName;
    String LogFileName;
    long RandomSeed;
    int NumberOfTrials;
    String Strategy;
    double DecisionThreshold;
    int DefectsPerTrial;
    boolean ReconfigureTestsPerTrial;
    boolean TestCostAsCoverage;

    int NumberOfModules;
    double[] CostPerModule;
    double SumCostOfAllModules;
    int[] TestsPerModule;
    double[] FailureRate;
```

```
int NumberOfTests;
double[] CostPerTest;
double SumCostOfAllTests;
int[] ModulesPerTest;
double[] CoveragePerModule;
//
//  ... derived attributes ...
//
Random Generator;
Environment env;
String experimentOutput = "";
//
//  ... simulation attributes ...
//
SystemObject system;
int CurrentTrialNumber = 0;
//
//  ... the FailureDeck for deciding which modules receive
//      defects at the start of a trial
//
private ArrayList<Module> FailureDeck = new ArrayList<Module>();

//
//  ... initialization method ...
//
public Configuration(Environment env) {

    this.env = env;

    this.CaseName = "default";
    this.LogFileName = "simulation.log";
    this.RandomSeed = 314159L;
    this.NumberOfTrials = 100;
    this.Strategy = "random";
    this.DecisionThreshold = 0.50;
    this.DefectsPerTrial = 1;
    this.ReconfigureTestsPerTrial = true;
    this.TestCostAsCoverage = false;

    this.NumberOfModules     = 20;
    this.CostPerModule       = new double [] {0.0, 1.0};
    this.SumCostOfAllModules = 1.0;
    this.TestsPerModule      = new int[] {1,4};
    this.FailureRate         = new double [] {0.1, 0.3};

    this.NumberOfTests     = 75;
    this.CostPerTest       = new double [] {0.0, 1.0};
    this.SumCostOfAllTests = 1.0;
    this.ModulesPerTest    = new int[] {1,4};
    this.CoveragePerModule = new double [] {0.1, 1.0};

    buildSystem();

}

//  ---------------
//  GET/SET Methods
//  ---------------

public SystemObject getSystemObject() {
   return this.system;
}

public String getExperimentOutputFile() {
    return this.experimentOutput;
}

public int getCurrentTrialNumber()      { return this.CurrentTrialNumber; }
```

```
    public void setCaseName(String x)       { this.CaseName = x; }
    public void setRandomSeed(long n)       { this.RandomSeed = n;}
    public void setNumberOfTrials(int n)    { this.NumberOfTrials = n; }
    public void setStrategy(String x)       { this.Strategy = x; }
    public void setDefectsPerTrial(int n)   { this.DefectsPerTrial = n; }
    public void setReconfigure(boolean b)   { this.ReconfigureTestsPerTrial = b; }

    public void setNumberOfModules(int n)   { this.NumberOfModules = n; }
    public void setMinCostPerModule(int n)  { this.CostPerModule[0] = n; }
    public void setMaxCostPerModule(int n)  { this.CostPerModule[1] = n; }
    public void setMinTestsPerModule(int n) { this.TestsPerModule[0] = n; }
    public void setMaxTestsPerModule(int n) { this.TestsPerModule[1] = n; }
    public void setMinFailureRate(double x) { this.FailureRate[0] = x; }
    public void setMaxFailureRate(double x) { this.FailureRate[1] = x; }

    public void setNumberOfTests(int n)     { this.NumberOfTests   = n; }
    public void setMinCostPerTest(double x) { this.CostPerTest[0] = x; }
    public void setMaxCostPerTest(double x) { this.CostPerTest[1] = x; }
    public void setMinModulesPerTest(int n) { this.ModulesPerTest[0] = n; }
    public void setMaxModulesPerTest(int n) { this.ModulesPerTest[1] = n; }
    public void setMinCoverage(double x)    { this.CoveragePerModule[0] = x; }
    public void setMaxCoverage(double x)    { this.CoveragePerModule[1] = x; }

    public String   getCaseName()        { return this.CaseName; }
    public long     getRandomSeed()      { return this.RandomSeed; }
    public int      getNumberOfTrials()  { return this.NumberOfTrials; }
    public String   getStrategy()        { return this.Strategy; }
    public int      getDefectsPerTrial() { return this.DefectsPerTrial; }
    public boolean  getReconfigure()     { return this.ReconfigureTestsPerTrial; }

    public int      getNumberOfModules() { return this.NumberOfModules; }
    public double[] getCostPerModule()   { return this.CostPerModule; }
    public int[]    getTestsPerModule()  { return this.TestsPerModule; }
    public double[] getFailureRate()     { return this.FailureRate; }

    public int      getNumberOfTests()   { return this.NumberOfTests; }
    public double[] getCostPerTest()     { return this.CostPerTest; }
    public int[]    getModulesPerTest()  { return this.ModulesPerTest; }
    public double[] getCoverage()        { return this.CoveragePerModule; }

    void copy (Configuration that) {

        this.CaseName = that.CaseName;
        this.LogFileName = that.LogFileName;
        this.RandomSeed = that.RandomSeed;
        this.NumberOfTrials = that.NumberOfTrials;
        this.Strategy = that.Strategy;
         this.DecisionThreshold = that.DecisionThreshold;
         this.DefectsPerTrial = that.DefectsPerTrial;
         this.ReconfigureTestsPerTrial =
             that.ReconfigureTestsPerTrial;
         this.TestCostAsCoverage = that.TestCostAsCoverage;

        this.NumberOfModules = that.NumberOfModules;

         this.CostPerModule = new double[2];
         this.CostPerModule[0] = that.CostPerModule[0];
         this.CostPerModule[1] = that.CostPerModule[1];

         this.SumCostOfAllModules = that.SumCostOfAllModules;

        this.TestsPerModule = new int[2];
        this.TestsPerModule[0] = that.TestsPerModule[0];
        this.TestsPerModule[1] = that.TestsPerModule[1];

         this.FailureRate = new double[2];
        this.FailureRate[0] = that.FailureRate[0];
        this.FailureRate[1] = that.FailureRate[1];
        this.NumberOfTests = that.NumberOfTests;

         this.CostPerTest = new double[2];
         this.CostPerTest[0] = that.CostPerTest[0];
         this.CostPerTest[1] = that.CostPerTest[1];

         this.SumCostOfAllTests = that.SumCostOfAllTests;

        this.ModulesPerTest = new int[2];
        this.ModulesPerTest[0] = that.ModulesPerTest[0];
        this.ModulesPerTest[1] = that.ModulesPerTest[1];

         this.CoveragePerModule = new double[2];
         this.CoveragePerModule[0] = that.CoveragePerModule[0];
         this.CoveragePerModule[1] = that.CoveragePerModule[1];

    } // end copy()


    void createFrom(Node node) {

        if ( node.getNodeType() != Node.ELEMENT_NODE ) return;

        Element elem = (Element) node;

        this.CaseName =
            Utility.updateStringItem(this.CaseName,"CaseName", elem);

        this.Strategy =
            Utility.updateStringItem(this.Strategy,"Strategy", elem);

        this.LogFileName =
            Utility.updateStringItem(this.LogFileName,"LogFileName", elem);

        this.RandomSeed =
            Utility.updateLongItem(this.RandomSeed,"RandomSeed", elem);
         //
         // ... if user specifies that the simulation should choose
         //     the random seed for the Generator, we fix this seed
         //     (using the current system time) so that multiple runs
         //     (or cases) will each use the same seed to make more
         //     useful comparisons
         //
         if ( this.RandomSeed < 0 ) {
           this.RandomSeed = Math.round(Math.random() * 1E6);
           env.logWrite("debug: RandomSeed set to %d",this.RandomSeed);
         }

        this.NumberOfTrials =
            Utility.updateIntItem(this.NumberOfTrials,"NumberOfTrials", elem);

         this.DecisionThreshold =
             Utility.updateDoubleItem(this.DecisionThreshold,
                                  "DecisionThreshold", elem);

         this.DefectsPerTrial =
             Utility.updateIntItem(this.DefectsPerTrial,"DefectsPerTrial",elem);

         this.ReconfigureTestsPerTrial =
             Utility.updateBooleanItem(this.ReconfigureTestsPerTrial,
                                     "ReconfigureTestsPerTrial", elem);

         this.TestCostAsCoverage =
             Utility.updateBooleanItem(this.TestCostAsCoverage,
                                     "TestCostAsCoverage", elem);

        this.NumberOfModules =
            Utility.updateIntItem(this.NumberOfModules,
                                 "NumberOfModules", elem);

         this.CostPerModule =
             Utility.updateDoubleMinMax(this.CostPerModule,
                                      "CostPerModule", elem);

         this.SumCostOfAllModules =
            Utility.updateDoubleItem(this.SumCostOfAllModules,
                                  "SumCostOfAllModules", elem);

        this.TestsPerModule =
            Utility.updateIntMinMax(this.TestsPerModule,
                                   "TestsPerModule", elem);

        this.FailureRate =
            Utility.updateDoubleMinMax(this.FailureRate,
                                     "FailureRate", elem);

        this.NumberOfTests =
            Utility.updateIntItem(this.NumberOfTests,
                                 "NumberOfTests", elem);

         this.CostPerTest =
            Utility.updateDoubleMinMax(this.CostPerTest,
                                     "CostPerTest", elem);

         this.SumCostOfAllTests =
            Utility.updateDoubleItem(this.SumCostOfAllTests,
                                  "SumCostOfAllTests", elem);

        this.ModulesPerTest =
            Utility.updateIntMinMax(this.ModulesPerTest,
                                  "ModulesPerTest", elem);

         this.CoveragePerModule =
             Utility.updateDoubleMinMax(this.CoveragePerModule,
                                        "CoveragePerModule", elem);
    } // end createFrom()


    public void readFromXML(String infile) {

      Document dom = null;

      File xmlfile = new File(infile);

      DocumentBuilderFactory dbf =
         DocumentBuilderFactory.newInstance();

      try {
         DocumentBuilder db = dbf.newDocumentBuilder();
         dom = db.parse(xmlfile.toURI().toString());
      }
      catch (ParserConfigurationException pce) {
         pce.printStackTrace();
      }
      catch (SAXException se) {
         se.printStackTrace();
      }
      catch (IOException ioe) {
         ioe.printStackTrace();
      }

      Element docEle = dom.getDocumentElement();

      NodeList cflist = docEle.getElementsByTagName("configuration");

      Node cfNode = cflist.item(0);
```

```
        this.createFrom(cfNode);

    }

    public void writeToXML(String outfile) {

        String xmlfile;

        if ( outfile.endsWith(".xml") ) {
            xmlfile = outfile;
        }
        else {
            xmlfile = outfile + ".xml";
        }

        try {
            PrintWriter output =
                new PrintWriter(new FileWriter(new File(xmlfile),false));

            output.printf("<?xml version=\"1.0\"?>\n");
            output.printf("<simulation>\n");
            output.printf(this.asXML());
            output.printf("</simulation>\n");
            output.close();
        }
        catch(IOException ioException) {
            System.err.println("Error writing XML file");
            System.exit(0);
        }

    } // end writeToXML();


    public void writeToXML() {
        String udr = env.getUserDataRoot();
        this.writeToXML(udr +"/"+ this.CaseName +"/configuration.xml");
    }


    public void report() {

        env.logWrite("Simulation Configuration");
        env.logWrite(".. Strategy .................. %s",
                         this.Strategy);
        env.logWrite(".. Random seed ............... %d",
                         this.RandomSeed);
         env.logWrite(".. Decision threshold ........ %f",
                         this.DecisionThreshold);
         env.logWrite(".. Number of trials .......... %d",
                         this.NumberOfTrials);
         env.logWrite(".. Defects per trial ......... %d",
                         this.DefectsPerTrial);

         env.logWrite(".. Reconfigure per trial .... "+
                          this.ReconfigureTestsPerTrial);

         env.logWrite(".. Test cost is coverage .... "+
                          this.TestCostAsCoverage);

        env.logWrite(".. Number of Modules ......... %d",
                         this.NumberOfModules);
         env.logWrite(".. Cost Per Module ........... %f %f",
                         this.CostPerModule[0],
                         this.CostPerModule[1]);
         env.logWrite(".. Sum Cost of All Modules ... %f",
                         this.SumCostOfAllModules);
        env.logWrite(".. TestsPerModule ............ %d %d",
                         this.TestsPerModule[0],
                         this.TestsPerModule[1]);
        env.logWrite(".. Failure Rate .............. %f %f",
                         this.FailureRate[0], this.FailureRate[1]);

        env.logWrite(".. Number of Tests ........... %d",
                         this.NumberOfTests);
         env.logWrite(".. Cost Per Test ............. %f %f",
                         this.CostPerTest[0],
                         this.CostPerTest[1]);
         env.logWrite(".. Sum Cost of All Tests ..... %f",
                         this.SumCostOfAllTests);
        env.logWrite(".. ModulesPerTest ............ %d %d",
                         this.ModulesPerTest[0],
                         this.ModulesPerTest[1]);
        env.logWrite(".. Coverage Per Module ....... %f %f",
                         this.CoveragePerModule[0],
                         this.CoveragePerModule[1]);

    } // end report()


    public void buildSystem() {

         //
         //  ... push configuration parameters
         //      to the log
         this.report();

        //
        //  ... compute derived attributes ...
        //

        if ( this.CaseName == "" ) {
            System.err.println("\nError in xml file");
            System.err.println(".. no CaseName found.\n");
            System.exit(0);
        }
        else {
             String udr = env.getUserDataRoot();
            File d = new File(udr + "/" + this.CaseName);
            if ( !d.exists() ) d.mkdirs();
        }

         if ( this.RandomSeed > 0 )
             this.Generator = new Random(this.RandomSeed);
         else
             this.Generator = new Random();

        int nModule    = this.NumberOfModules;
        int nTest      = this.NumberOfTests;

        this.system = new SystemObject(this.env);
        SystemObject s = this.system;

         double lorate = this.FailureRate[0];
         double hirate = this.FailureRate[1];

         double rawSumModuleCost = 0.0;
        for (int i = 0; i < nModule; i++) {
            String name = String.format("M%03d",i);
            double frate =
                lorate + ( hirate - lorate)*this.Generator.nextDouble();
            Module m = new Module(name,frate);
             double locost = this.CostPerTest[0];
             double hicost = this.CostPerTest[1];
            double cost  =
                locost + (hicost - locost)*this.Generator.nextDouble();
             rawSumModuleCost += cost;
             m.setCost(cost);
            s.addModule(m);
        }

         double rawSumTestCost = 0.0;
        for (int i = 0; i < nTest; i++) {
            String name = String.format("T%03d",i);
            double frate = this.Generator.nextDouble();
             double locost = this.CostPerTest[0];
             double hicost = this.CostPerTest[1];
            double cost  =
                locost + (hicost - locost) * this.Generator.nextDouble();
             rawSumTestCost += cost;
            Test t = new Test(name);
            t.setCost(cost);
            s.addTest(t);
        }

         for (int i=0; i < nModule; i++) {
            Module m = s.getModule(i);
             double p = m.getCost();
             p = p * (this.SumCostOfAllModules / rawSumModuleCost );
             m.setCost(p);
         }

         for (int i=0; i < nTest; i++) {
            Test t = s.getTest(i);
             double p = t.getCost();
             p = p * (this.SumCostOfAllTests / rawSumTestCost );
             t.setCost(p);
         }

        configureTests();

         system.describeModules();
         // system.describeTests();

    } // end buildSystem()


    void configureTests() {

        SystemObject s = this.system;

        int[] ModuleDeck = generateRandomList(s.getModuleCount());

        int[] TestDeck   = generateRandomList(s.getTestCount());

        int nModule = s.getModuleCount();

        int nTest   = s.getTestCount();

        //
        //  ... remove existing probes ...
        //
        s.removeAllProbes();

         //
         //  ... establish minimum and maximum coverage
         //
         double loalpha = CoveragePerModule[0];
         double hialpha = CoveragePerModule[1];

         //
         //  ... create a random bit matrix
         //
         long bmseed = this.Generator.nextLong();
         BitMatrix bm = new BitMatrix(nModule, nTest,bmseed);
         int lotpm = TestsPerModule[0];
         int hitpm = TestsPerModule[1];
         int lompt = ModulesPerTest[0];
```

```
       int himpt = ModulesPerTest[1];
        bm.randomize(lotpm,hitpm,lompt,himpt);
        env.logWrite("Actual matrix row range    = [%d,%d]",
                     bm.getMinRowCount(), bm.getMaxRowCount());
        env.logWrite("Actual matrix column range = [%d,%d]",
                     bm.getMinColumnCount(), bm.getMaxColumnCount());
        String mat = bm.graphic();
        env.logWrite("\n"+mat);

        int [] match = bm.getNonZeroCells();
        for (int k=0; k < match.length; k += 2) {
            int mi = match[k];
            int ti = match[k+1];
            Module md = s.getModule(ModuleDeck[mi]);
            Test t = s.getTest(TestDeck[ti]);
            double cp = Generator.nextDouble();
            double fr =
                loalpha + (hialpha - loalpha)*Generator.nextDouble();
            t.addProbe(md,cp,fr);
        }

        //
        // ... kdp ... set cost by coverage
        //
        if ( this.TestCostAsCoverage ) {
          for (int k=0; k < nTest; k++) {
            Test t = s.getTest(k);
            t.setCostByCoverage();
          }
        }

    } // end configureTests()



    //
    //  ... method: create Failure Deck used in deciding in which
    //              modules the simulation will plant defects
    //
    void createFailureDeck(SystemObject s, int nTrial) {
        //
        //  ... find the min and max
        //      of component reliabilities
        //
        int nModule = s.getModuleCount();
        double min = 1.0;
        double max = 0.0;
        double sum = 0.0;
        for (int i = 0; i < nModule; i++) {
            Module md = s.getModule(i);
            sum += md.getFailureRate();
            if ( md.getFailureRate() > max )
                max = md.getFailureRate();
            if ( md.getFailureRate() < min )
                min = md.getFailureRate();
        }
        //
        //  ... populate the deck ...
        //
        for (int i = 0; i < nModule; i++) {
            Module md = s.getModule(i);
            int factor =
                (int) ((md.getFailureRate() / sum) * nTrial + 0.5);
            env.logWrite(".. Failure Deck: %s(%e) appears %5d times",
                         md.getName(), md.getFailureRate(), factor);
            for (int j = 0; j < factor; j++) {
                FailureDeck.add(md);
            }
        }
        //
        //  ... shuffle the deck ...
        //
        int nDeck = FailureDeck.size();
        env.logWrite(".. Failure Deck has %5d entries ..",nDeck);
        for (int i = 0; i < nDeck; i++) {
            int p = this.Generator.nextInt(nDeck);
            int q = this.Generator.nextInt(nDeck);
            Module mx = FailureDeck.get(p);
            Module my = FailureDeck.get(q);
            FailureDeck.set(p,my);
            FailureDeck.set(q,mx);
        }
        //
        //  ... done ...
        //
    }

    //
    //  ----------------------
    //  TEST SELECTION METHODS
    //  ----------------------
    //
    //    All methods return a list (array) of tests
    //    though in many cases these are singleton lists
    //

    //
    //  ... method: select next test at random
    //              this method returns ALL tests
    //              in the Test list, randomized
    //
    //      to help with reproducibility from run
    //      to run, a local Random() generator is
    //      created, separate from the Configuration
    //      generator
    //
    Test[] getRandomNextTest(ArrayList<Test> tlist) {

        int nTest = tlist.size();

        Test[] tmax = new Test[nTest];


        long seed = Math.round(1.0E6*Math.random());
        Random rnd = new Random(seed);

        for (int i=0; i < nTest; i++)
          tmax[i] = tlist.get(i);

        for (int i=0; i < nTest; i++) {
          int q = rnd.nextInt(nTest);
          int p = rnd.nextInt(nTest);
          Test t = tmax[p];
          tmax[p] = tmax[q];
          tmax[q] = t;
        }

        return tmax;

    } // end getRandomNextTest()

    //
    //  ... method: select next best test based on
    //              forecast entropy reduction
    //
    Test[] getBestNextTest(SystemObject s, ArrayList<Test> tlist) {
        Test[] tmax = new Test[1];
        double dmax = -9999.0;
```

```
         int nTest = tlist.size();
        int imax = 0;
        for (int i=0; i < nTest; i++) {
            Test t = tlist.get(i);
            double dh = Utility.deltaEntropy(s,t);
            // double dh = Utility.deltaMaxP(s,t);
            // double dh = Utility.deltaKL(s,t);
            double dc = t.getCost();
            double df = dh / dc;
            if ( df > dmax ) {
               dmax = df;
               imax = i;
            }
        }
        tmax[0] = tlist.get(imax);
        return tmax;

    } // end getBestNextTest()

    //
    //  ... method: select next best test based on
    //              forecast increase in MaxP
    //
    Test[] getBestNextTestMaxP(SystemObject s, ArrayList<Test> tlist) {
         Test[] tmax = new Test[1];
        double dmax = -9999.0;
         int nTest = tlist.size();
        int imax = 0;
        for (int i=0; i < nTest; i++) {
            Test t = tlist.get(i);
            double dh = Utility.deltaMaxP(s,t);
            // double dh = Utility.deltaMaxP(s,t);
            // double dh = Utility.deltaKL(s,t);
            double dc = t.getCost();
            double df = dh / dc;
            if ( df > dmax ) {
               dmax = df;
               imax = i;
            }
        }
        tmax[0] = tlist.get(imax);
        return tmax;

    } // end getBestNextTestMaxP()


    //
    //  ... method: select next best TWO tests
    //
    Test[] getBestNextTwoTests(SystemObject s, ArrayList<Test> tlist) {

        double dmax = -9999.0;
         int nTest = tlist.size();

         //
         //  ... if only one test left, it is by default our
         //      best next test
         //
         if ( nTest == 1 ) {
           Test[] tmax = new Test[1];
           tmax[0] = tlist.get(0);
           return tmax;
         }

         Test tmax1 = tlist.get(0);
         Test tmax2 = tlist.get(1);

         //
         // ... baseline entropy ...
         //
         double s0 = s.Entropy();

         for (int i=0; i < nTest; i++) {

           Test t1 = tlist.get(i);

           //
           //  ... T1=PASS ...
           //
           s.pushState();
           Utility.updateOnTestAsPASS(s,t1);
           //
           //  ... T1=PASS, T2=PASS ...
           //
           for (int j=0; j < nTest; j++) {
             //
             // ... do not check test with itself ...
             //
             if ( i != j ) {
               Test t2 = tlist.get(j);
               s.pushState();
               Utility.updateOnTestAsPASS(s,t2);
               double dh =
                 (s0 - s.Entropy())*t1.probabilityPass()*t2.probabilityPass();
               double dc = (t1.getCost() + t2.getCost());
               double df = dh / dc;
               if ( df > dmax ) {
                 dmax = df;
                 tmax1 = t1;
                 tmax2 = t2;

               } // end if(df > dmax)
               //
               //  ... undo application of T2=PASS ...
               //
               s.popState();
             } // end if (i != j)
           } // end for(j)
           //
           //  ... T1=PASS, T2=FAIL ...
           //
           for (int j=0; j < nTest; j++) {
             //
             // ... do not check test with itself ...
             //
             if ( i != j ) {
               Test t2 = tlist.get(j);
               s.pushState();
               Utility.updateOnTestAsFAIL(s,t2);
               double dh =
                 (s0 - s.Entropy())*t1.probabilityPass()*t2.probabilityFail();
               double dc = (t1.getCost() + t2.getCost());
               double df = dh / dc;
               if ( df > dmax ) {
                 dmax = df;
                 tmax1 = t1;
                 tmax2 = t2;
               } // end if(df > dmax)
               //
               //  ... undo application of T2=FAIL ...
               //
               s.popState();
             } // end if (i != j)
           } // end for(j)
           //
           //  ... undo application of T1=PASS ...
           //
           s.popState();
           //
           //  ... T1=FAIL ...
           //
           s.pushState();
           Utility.updateOnTestAsFAIL(s,t1);
           //
           //  ... T1=FAIL, T2=PASS ...
           //
           for (int j=0; j < nTest; j++) {
             //
             // ... do not check test with itself ...
             //
             if ( i != j ) {
               Test t2 = tlist.get(j);
               s.pushState();
               Utility.updateOnTestAsPASS(s,t2);
               double dh =
                 (s0 - s.Entropy())*t1.probabilityFail()*t2.probabilityPass();
               double dc = (t1.getCost() + t2.getCost());
               double df = dh / dc;
               if ( df > dmax ) {
                 dmax = df;
                 tmax1 = t1;
                 tmax2 = t2;

               } // end if(df > dmax)
               //
               //  ... undo application of T2=PASS ...
               //
               s.popState();
             } // end if (i != j)
           } // end for(j)
           //
           //  ... T1=FAIL, T2=FAIL ...
           //
           for (int j=0; j < nTest; j++) {
             //
             // ... do not check test with itself ...
             //
             if ( i != j ) {
               Test t2 = tlist.get(j);
               s.pushState();
               Utility.updateOnTestAsFAIL(s,t2);
               double dh =
                 (s0 - s.Entropy())*t1.probabilityFail()*t2.probabilityFail();
               double dc = (t1.getCost() + t2.getCost());
               double df = dh / dc;
               if ( df > dmax ) {
                 dmax = df;
                 tmax1 = t1;
                 tmax2 = t2;
               } // end if(df > dmax)
               //
               //  ... undo application of T2=FAIL ...
               //
               s.popState();
             } // end if (i != j)
           } // end for(j)
           //
           //  ... undo application of T1=FAIL ...
           //
           s.popState();

         } // end for(i)

         Test[] tmax = new Test[2];
         tmax[0] = tmax1;
         tmax[1] = tmax2;
         env.logWrite("...");
```

```
       env.logWrite("... BEST TWO TESTS: %s and %s",
                tmax1.getName(), tmax2.getName());
        env.logWrite("...");
        return tmax;

    } // end getBestNextTwoTests()

    //
    //  ... method: select next best TWO tests
    //
    Test[] getBestNextTwoTestsMaxP(SystemObject s, ArrayList<Test> tlist) {

       double dmax = -9999.0;
        int nTest = tlist.size();

        //
        //  ... if only one test left, it is by default our
        //      best next test
        //
        if ( nTest == 1 ) {
          Test[] tmax = new Test[1];
          tmax[0] = tlist.get(0);
          return tmax;
        }

        Test tmax1 = tlist.get(0);
        Test tmax2 = tlist.get(1);

        //
        // ... baseline entropy ...
        //
        double s0 = s.MaxP();

        for (int i=0; i < nTest; i++) {

          Test t1 = tlist.get(i);

          //
          //  ... T1=PASS ...
          //
          s.pushState();
          Utility.updateOnTestAsPASS(s,t1);
          //
          //  ... T1=PASS, T2=PASS ...
          //
          for (int j=0; j < nTest; j++) {
            //
            // ... do not check test with itself ...
            //
            if ( i != j ) {
              Test t2 = tlist.get(j);
              s.pushState();
              Utility.updateOnTestAsPASS(s,t2);
              double dh =
                (s.MaxP() - s0)*t1.probabilityPass()*t2.probabilityPass();
              double dc = (t1.getCost() + t2.getCost());
              double df = dh / dc;
              if ( df > dmax ) {
                dmax = df;
                tmax1 = t1;
                tmax2 = t2;

              } // end if(df > dmax)
              //
              //  ... undo application of T2=PASS ...
              //
              s.popState();
            } // end if (i != j)
          } // end for(j)
          //
          //  ... T1=PASS, T2=FAIL ...
          //
          for (int j=0; j < nTest; j++) {
            //
            // ... do not check test with itself ...
            //
            if ( i != j ) {
              Test t2 = tlist.get(j);
              s.pushState();
              Utility.updateOnTestAsFAIL(s,t2);
              double dh =
                (s.MaxP() - s0)*t1.probabilityPass()*t2.probabilityFail();
              double dc = (t1.getCost() + t2.getCost());
              double df = dh / dc;
              if ( df > dmax ) {
                dmax = df;
                tmax1 = t1;
                tmax2 = t2;
              } // end if(df > dmax)
              //
              //  ... undo application of T2=FAIL ...
              //
              s.popState();
            } // end if (i != j)
          } // end for(j)
          //
          //  ... undo application of T1=PASS ...
          //
          s.popState();
          //
          //  ... T1=FAIL ...
          //
          s.pushState();
          Utility.updateOnTestAsFAIL(s,t1);
          //
          //  ... T1=FAIL, T2=PASS ...
          //
          for (int j=0; j < nTest; j++) {
            //
            // ... do not check test with itself ...
            //
            if ( i != j ) {
              Test t2 = tlist.get(j);
              s.pushState();
              Utility.updateOnTestAsPASS(s,t2);
              double dh =
                (s.MaxP() - s0)*t1.probabilityFail()*t2.probabilityPass();
              double dc = (t1.getCost() + t2.getCost());
              double df = dh / dc;
              if ( df > dmax ) {
                dmax = df;
                tmax1 = t1;
                tmax2 = t2;

              } // end if(df > dmax)
              //
              //  ... undo application of T2=PASS ...
              //
              s.popState();
            } // end if (i != j)
          } // end for(j)
          //
          //  ... T1=FAIL, T2=FAIL ...
          //
          for (int j=0; j < nTest; j++) {
            //
            // ... do not check test with itself ...
            //
            if ( i != j ) {
              Test t2 = tlist.get(j);
              s.pushState();
              Utility.updateOnTestAsFAIL(s,t2);
              double dh =
                (s.MaxP() - s0)*t1.probabilityFail()*t2.probabilityFail();
              double dc = (t1.getCost() + t2.getCost());
              double df = dh / dc;
              if ( df > dmax ) {
                dmax = df;
                tmax1 = t1;
                tmax2 = t2;
              } // end if(df > dmax)
              //
              //  ... undo application of T2=FAIL ...
              //
              s.popState();
            } // end if (i != j)
          } // end for(j)
          //
          //  ... undo application of T1=FAIL ...
          //
          s.popState();

        } // end for(i)

        Test[] tmax = new Test[2];
        tmax[0] = tmax1;
        tmax[1] = tmax2;
        env.logWrite("...");
       env.logWrite("... BEST TWO MAXP TESTS: %s and %s",
              tmax1.getName(), tmax2.getName());
        env.logWrite("...");
        return tmax;

    } // end getBestNextTwoTests()



    //
    //  ... method: select worst next test based on
    //              increase in entropy
    //
    Test[] getWorstNextTest(SystemObject s, ArrayList<Test> tlist) {
        Test[] tmin = new Test[1];
       double dmin = 9999.0;
        int nTest = tlist.size();
       int imin = 0;
       for (int i=0; i < tlist.size(); i++) {
           Test t = tlist.get(i);
           double dh = Utility.deltaEntropy(s,t);
           double dc = t.getCost();
           double df = dh / dc;
           if ( df < dmin ) {
              dmin = df;
              imin = i;
           }
       }
       tmin[0] = tlist.get(imin);
       return tmin;
    }

    //
    //  ... method: plant defect in module selected at random
    //              from Failure Deck, at a point selected at
    //              random on the interval [0,1]
    //
    int plantDefect(SystemObject s) {
        int nTrial = this.NumberOfTrials;

       if ( FailureDeck.size() == 0 )
           createFailureDeck(s,nTrial);

       Module mi = FailureDeck.get(0);
       FailureDeck.remove(0);

       double defect = this.Generator.nextDouble();
        // ... experiment ...
        // double defect = 0.0;

       mi.setDefectAt(defect);

       int imod = s.getModuleIndex(mi);

       env.logWrite("... Setting defect at %f in module %s",
              defect, mi.getName());

       return imod;

    }



    //
    //  ... utility method, generate a random list of nSize elements
    //
    int[] generateRandomList ( int nSize ) {
       int [] result = new int[nSize];
       for (int i = 0; i < nSize; i++)
           result[i] = i;
       for (int i = 0; i < nSize; i++) {
           int p = this.Generator.nextInt(nSize);
           int q = this.Generator.nextInt(nSize);
           int r = result[p];
           result[p] = result[q];
           result[q] = r;
       }
       return result;
    }



    String asXML() {

       String result = "";

       result += "<configuration>\n";

       result += Utility.tagXML(this.CaseName, "CaseName");
       result += Utility.tagXML(this.Strategy, "Strategy");
       result += Utility.tagXML(this.RandomSeed, "RandomSeed");
       result += Utility.tagXML(this.NumberOfTrials,"NumberOfTrials");
       result += Utility.tagXML(this.DecisionThreshold, "DecisionThreshold");
       result += Utility.tagXML(this.DefectsPerTrial, "DefectsPerTrial");
       result += Utility.tagXML(this.NumberOfModules, "NumberOfModules");
       result += "<CostPerModule>\n";
       result += "  " + Utility.tagXML(this.CostPerModule[0],"Minimum");
       result += "  " + Utility.tagXML(this.CostPerModule[1],"Maximum");
       result += "</CostPerModule>\n";
       result +=
        Utility.tagXML(this.SumCostOfAllModules, "SumCostOfAllModules");
       result += "<FailureRate>\n";
       result += "  " + Utility.tagXML(this.FailureRate[0],"Minimum");
       result += "  " + Utility.tagXML(this.FailureRate[1],"Maximum");
       result += "</FailureRate>\n";
       result += Utility.tagXML(this.NumberOfTests, "NumberOfTests");
       result += "<CostPerTest>\n";
```

```
        result += "  " + Utility.tagXML(this.CostPerTest[0],"Minimum");
        result += "  " + Utility.tagXML(this.CostPerTest[1],"Maximum");
        result += "</CostPerTest>\n";
        result +=
          Utility.tagXML(this.SumCostOfAllTests, "SumCostOfAllTests");
        result += "<ModulesPerTest>\n";
        result += "  " + Utility.tagXML(this.ModulesPerTest[0],"Minimum");
        result += "  " + Utility.tagXML(this.ModulesPerTest[1],"Maximum");
        result += "</ModulesPerTest>\n";
        result += "<TestsPerModule>\n";
        result += "  " + Utility.tagXML(this.TestsPerModule[0],"Minimum");
        result += "  " + Utility.tagXML(this.TestsPerModule[1],"Maximum");
        result += "</TestsPerModule>\n";
        result += "<CoveragePerModule>\n";
        result += "  " + Utility.tagXML(this.CoveragePerModule[0],"Minimum");
        result += "  " + Utility.tagXML(this.CoveragePerModule[1],"Maximum");
        result += "</CoveragePerModule>\n";
        result += "</configuration>\n";

        return result;

    } // end asXML()


    //
    //  method: execute
    //  ---------------
    //
    public void execute() {

        buildSystem();

        int nModule = this.NumberOfModules;
        int nTest   = this.NumberOfTests;
        int nTrial  = this.NumberOfTrials;
         double thresh = this.DecisionThreshold;

         String udr = env.getUserDataRoot();
        String outdir = udr + "/" + this.CaseName;


         writeToXML(outdir + "/configuration.xml");

         this.experimentOutput =  outdir + "/experiment.xml";
         Utility.removeFile(experimentOutput);
         Utility.writeToFile(experimentOutput, "<?xml version=\"1.0\" ?>\n");
         Utility.writeToFile(experimentOutput, "<experiment>\n");

         String sysoutdir = outdir + "/systemobject";
         Utility.deleteDirectory(new File(sysoutdir));
        File d = new File(sysoutdir);
        if ( !d.exists() ) d.mkdirs();

        this.report();

        env.logWrite("================");
        env.logWrite("... Starting ...");
        env.logWrite("================");

        SystemObject s = this.system;

        s.describeModules();

        s.describeTests();

        env.logWrite("System entropy = %f, distance = %f",
                  s.Entropy(),s.Distance());

         double[][] testcost = new double[nTrial][nTest + 1];
         long[] timing   = new long[nTrial];

        double h0    = s.Entropy();

        for (int i=0; i < nTrial; i++) {
             testcost[i][0] = 0.0;
        }

        ArrayList<Test> TestList;

        //
        //  --------------------
        //  MAIN SIMULATION LOOP
        //  --------------------
        //
        for (int j=0; j < nTrial; j++) {

             this.CurrentTrialNumber = j;

             long startTime = System.currentTimeMillis();

             String outstr = String.format("<trial id=\"%05d\">",j);
            env.logWrite(outstr);

             Utility.writeToFile(experimentOutput, outstr + "\n");

             // Utility.writeToFile(sysout, outstr + "\n");
             // Utility.writeToFile(sysout, s.describeTestsAsXML());
             // Utility.writeToFile(sysout, "</trial>\n");
             this.system.
               writeToXML(sysoutdir+String.format("/trial_%05d.xml",j));

             double sumCostTest = 0.0;

             //
             //  ... dump the initial state of the system
             //      which should be the prior probabilities
             //      for all modules
             //
             Utility.writeToFile(experimentOutput, s.dumpStateAsXML());

            //
            //  ... at the start of a simulation trial
            //      plant defect(s) in System
            //
             while ( s.getDefectCount() < this.DefectsPerTrial ) {
               int imod = plantDefect(s);
             }

            //
            //  ... following the defect planting, dump the
            //      true state of the system to file
            //
             Utility.writeToFile(experimentOutput, s.dumpTrueStateAsXML());

            //
            //  ... create a scratch pad Test list that we can
            //      consume as testing proceeds (no test should
            //      be run twice)
            //
            TestList = s.copyTestList();

            int i = 0;

            //
            //  -----------------
            //  MAIN TESTING LOOP
            //  -----------------
            //
            while ( TestList.size() > 0 ) {


                //
                //  ... select test(s) to apply based on
                //      the requested strategy
                //

                Test[] tapply = new Test[0];

                if ( this.Strategy.equals("random") ) {
                    tapply = getRandomNextTest(TestList);
                 }
                else if ( this.Strategy.equals("best") ) {
                    tapply = getBestNextTest(s,TestList);
                 }
                else if ( this.Strategy.equals("bestmaxp") ) {
                    tapply = getBestNextTestMaxP(s,TestList);
                 }
                 else if ( this.Strategy.equals("best2") ) {
                     tapply = getBestNextTwoTests(s,TestList);
                 }
                else if ( this.Strategy.equals("best2maxp") ) {
                    tapply = getBestNextTwoTestsMaxP(s,TestList);
                }
                else if ( this.Strategy.equals("worst") ) {
                    tapply = getWorstNextTest(s,TestList);
                 }
                 else {
                   System.err.printf("\nERROR: unknown strategy %s\n\n",
                     this.Strategy);
                   System.exit(1);
                 }

                 int nSelected = tapply.length;


                 for (int k = 0; k < nSelected; k++) {

                   sumCostTest += tapply[k].getCost();

                   env.logWrite(".. (%03d) Test=%s delta H = %f sum Cost = %f",
                           i,
                         tapply[k].getName(),
                         Utility.deltaEntropy(s,tapply[k]),
                           sumCostTest);

                   //
                   //  ... Apply test to the system ...
                  //
                  Utility.updateOnTest(s,tapply[k]);
                  //
                  //  ... After running (applying) test,
                  //      dump the measured state of the system
                   //
                   Utility.writeToFile(experimentOutput, s.dumpStateAsXML());
                   //
                   //  ... Then, remove test from the scratch pad list
                  //
                   Test tdel = tapply[k];
                   TestList.remove(TestList.lastIndexOf(tdel));

                  double h = s.Entropy();
                  env.logWrite("... Entropy = %f, Distance = %f",
                         h,s.Distance());

                  i++;
                   testcost[j][i] = sumCostTest;
                } // end k-loop applying tests

            }
             //
            //  -------------------
            //  END OF TESTING LOOP
            //  -------------------
             //
             Utility.writeToFile(experimentOutput, "</trial>\n");

            s.Reset();

             if ( this.ReconfigureTestsPerTrial ) {
                 configureTests();
                 s.describeTests();
             }

             timing[j] = System.currentTimeMillis() - startTime;

        } // end for(j)
         //
         //  ----------------
         //  END OF MAIN LOOP
         //  ----------------
         //

         Utility.writeToFile(experimentOutput, "</experiment>\n");
         // Utility.writeToFile(sysout, "</systemobject>\n");


         double[][][] trialData =
            Utility.getTrialDataFromFile(experimentOutput);

         double[] meanH = Utility.computeMeanEntropyByTrial(trialData);
         double[] meanP = Utility.computeMeanMaxPByTrial(trialData);

         Utility.dumpArrayToFile(meanH, outdir, "mean-entropy.dat");
         Utility.dumpArrayToFile(meanP, outdir, "mean-maxp.dat");

         double[] meanCost     = Utility.computeMeanByTrial(testcost);

         Utility.dumpArrayToFile(meanCost, outdir, "testcost.dat");

         Utility.dumpArrayToFile(timing, outdir, "timing.dat");
         Utility.writeToFile(outdir + "/timing.dat",
           String.format("# Mean: %f\n",
           Utility.computeMeanByTrial(timing)));
         Utility.writeToFile(outdir + "/timing.dat",
           String.format("# Var: %f\n",
           Utility.computeVarianceByTrial(timing)));

        env.logWrite("Observed Failures ...");
        s.sortModulesByFailureRate();
        for (int i=0; i < nModule; i++) {
            Module md = s.getModule(i);
            env.logWrite("... %s (%e) failed %5d times",
                      md.getName(),
                      md.getFailureRate(),
                      md.getFailureCount());
        }

        env.logWrite("============");
        env.logWrite("... DONE ...");
        env.logWrite("============");

    } // end method execute()
```

```
}
//
//  ========================
//  end class: Configuration
//  ========================
//
```

B.2 SystemObject.java

```
//
//  =====================================================================
//
//  Module:  SystemObject.java
//
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2011
//  =====================================================================
//
package rbts.obj;

import rbts.common.*;


import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.xml.sax.SAXException;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.io.*;
//
//  package: SystemObject.java
//  --------------------------
//
//    The SystemObject is the model for the system and container
//    for Module and Test objects.
//
//    This theoretical System is comprised of a collection of
//    Modules with known (or estimated) failure rates, and a
//    collection of Tests, each of which exercises one or more
//    Modules within the system.
//
public class SystemObject {

    //
    //  ... attributes ...
    //
    ArrayList<Module> ModuleList   = new ArrayList<Module>();
    ArrayList<Test> TestList       = new ArrayList<Test>();
    ArrayList<double[]> StateStack = new ArrayList<double[]>();

    Environment env;

    //
    //  ... methods ...
    //

    public SystemObject(Environment env) {
      this.env = env;
    }

    public Environment getEnv() {
      return this.env;
    }
```

```
public void addModule(Module m) {
   this.ModuleList.add(m);
}

public void addModuleList(Module[] mlist) {
  for (int i=0; i < mlist.length; i++) {
    this.ModuleList.add(mlist[i]);
  }
}

public Module getModule(int q) {
   return this.ModuleList.get(q);
}


public int getModuleCount() {
    return this.ModuleList.size();
}

public int getDefectCount() {
    int count = 0;
    int nModule = this.ModuleList.size();
    for (int i = 0; i < nModule; i++)
      if ( this.ModuleList.get(i).hasDefect() ) count++;
    return count;
}

public Module getModuleByName(String name) {
  for (int i = 0; i < this.ModuleList.size(); i++) {
    Module m = this.ModuleList.get(i);
    String current = m.getName();
    if ( current.equals(name) ) return m;
  }
  return null;
}

public int getModuleIndex(Module m) {
   int i;
   for (i = 0; i < this.ModuleList.size(); i++) {
       if ( this.ModuleList.get(i) == m ) return i;
   }
   return -1;
}

public void addTest(Test t) {
   this.TestList.add(t);
}

public void addTestList(Test[] tlist) {
  for (int i=0; i < tlist.length; i++) {
    this.TestList.add(tlist[i]);
  }
}

public Test getTest(int q) {
   return this.TestList.get(q);
}

public int getTestIndex(Test t) {
   for (int i = 0; i < this.TestList.size(); i++) {
       if ( this.TestList.get(i) == t ) return i;
   }
   return -1;
}

public int getTestCount() {
    return this.TestList.size();
}
```

```
public ArrayList<Test> copyTestList() {
  ArrayList<Test> tlist = new ArrayList<Test>();
  ArrayList<Test> olist = this.TestList;
  for (int i=0; i < olist.size(); i++)
    tlist.add(olist.get(i));
  return tlist;
}

public void removeAllProbes() {

   int nModule = this.ModuleList.size();
   int nTest = this.TestList.size();

   for (int i=0; i < nModule; i++) {
       Module md = this.ModuleList.get(i);
        md.removeAllProbes();
   }

   for (int i=0; i < nTest; i++) {
       Test t = this.TestList.get(i);
        t.removeAllProbes();
   }

   env.logWrite("... ALL PROBES REMOVED ...");

}

public double MaxProbability() {
  double maxp = 0.0;
  int nModule = this.ModuleList.size();
  for (int i=0; i < nModule; i++) {
    double p = this.getModule(i).getProbabilityBad();
    maxp += Math.max(p, 1-p);
  }
  maxp /= this.ModuleList.size();
  return maxp;
}

public double MaxP() {
  return this.MaxProbability();
}

public double DeltaMaxP() {
  double pdel = 0.0;
  int nModule = this.ModuleList.size();
  for (int i=0; i < nModule; i++) {
    double p = this.getModule(i).getProbabilityBad();
    pdel += ( Math.max(p, 1-p) - 0.5);
  }
  pdel /= this.ModuleList.size();
  return pdel;
}


public double Entropy() {
  double esum = 0.0;
  int nModule = this.ModuleList.size();
  for (int i = 0; i < nModule; i++) {
      double bi = this.ModuleList.get(i).getProbabilityBad();
      esum += Utility.entropy(bi);
  }
  esum /= nModule;
  return esum;
}

public double[] getTrueStateVector() {
  int nModule = this.getModuleCount();
  double[] trueState = new double[nModule];
```

```
      for (int i=0; i < nModule; i++) {
        trueState[i] = 0.0;
        if ( this.getModule(i).hasDefect() )
          trueState[i] = 1.0;
      }
      return trueState;
    }

    public double[] getStateVector() {
      int nModule = this.getModuleCount();
      double[] actualState = new double[nModule];
      for (int i=0; i < nModule; i++) {
        actualState[i] = this.getModule(i).getProbabilityBad();
      }
      return actualState;
    }

    public double Distance() {
      int nModule = this.getModuleCount();
      double[] trueState = this.getTrueStateVector();
      double sum = 0.0;
      for (int i=0; i < nModule; i++) {
        double dx = trueState[i] - this.getModule(i).getProbabilityBad();
        sum += dx*dx;
      }
      return Math.sqrt(sum);
    }

    public double KullbackLeiblerDivergence() {
      int nModule = this.getModuleCount();
      double[] trueState = this.getTrueStateVector();
      double sum = 0.0;
      for (int i=0; i < nModule; i++) {
        double ai = trueState[i];
        double bi = this.getModule(i).getProbabilityBad();
        double dx = Utility.kullbackleibler(ai,bi);
        sum += dx;
      }
      return sum;

    }

    public int CorrectReplacementDecisions(double threshold) {
      int nModule = this.getModuleCount();
      double[] trueState = this.getTrueStateVector();
      int count = 0;
      for (int i = 0; i < nModule; i++) {
        double bi = this.getModule(i).getProbabilityBad();
        int decision = 0;
        if ( bi >= threshold ) decision = 1;
        if ( decision == trueState[i] ) count++;
      }
      return count;
    }

    public void describeTests() {

        double sumCost = 0.0;
       int nTest = this.TestList.size();
       int i,j;
       for (i = 0; i < nTest; i++) {
           Test t = this.TestList.get(i);
            sumCost += t.getCost();
           env.logWrite("Test %s, Cost(%f) Pj(%f)",
              t.getName(),t.getCost(),t.probabilityPass());
           int nProbe = t.ProbeList.size();
           for (j = 0; j < nProbe; j++) {
               Probe p = t.ProbeList.get(j);
               Module m = p.getModule();
```

```
                env.logWrite("   Module %s, coverage = %f",
                                 m.getName(), p.getFraction());
            }
        }
        env.logWrite(".. TOTAL COST OF ALL TESTS ....... %f",sumCost);
    }

    public void describeModules() {

        double sumCost = 0.0;
        int nModule = this.ModuleList.size();
        int nTest = this.TestList.size();
        int i,j,k;
        for (i = 0; i < nModule; i++) {
            Module m = this.ModuleList.get(i);
            sumCost += m.getCost();
            env.logWrite("Module %s: Failure rate(%8.2e) H(%8.2e)",
                         m.getName(),m.getFailureRate(),
                          Utility.entropy(m.getProbabilityBad()));
            int nProbe = m.getProbeCount();
            for (j = 0; j < nProbe; j++) {
                Probe pb = m.getProbe(j);
                Test t = pb.getTest();
                int nGraphic = 30;
                env.logWrite("  Test %s (%5.2f) %s",
                                 t.getName(),
                                 pb.getFraction(),
                                 pb.getCoverageGraphic(nGraphic));
            } // end for(j)
        } // end for(i)
        env.logWrite(".. TOTAL COST OF ALL MODULES ..... %f",sumCost);
    } // end describeModules()


    public void dumpTrueStateToFile(String outdir, String outfile) {
      File d = new File(outdir);
      if ( !d.exists() ) d.mkdirs();
      this.dumpTrueStateToFile(outdir + "/" + outfile);
    }


    public void dumpTrueStateToFile(String outfile) {
      try {
        PrintWriter output =
          new PrintWriter(new FileWriter(new File(outfile)));
        double[] trueState = this.getTrueStateVector();
        for (int i = 0; i < trueState.length; i++) {
          output.printf("%f\n",trueState[i]);
        }
        output.close();
      }
      catch(IOException ioException) {
        System.out.println("Error writing true state to file");
        System.exit(0);
      }
    }


    public void dumpStateToFile(String outdir, String outfile) {
      File d = new File(outdir);
      if ( !d.exists() ) d.mkdirs();
      this.dumpStateToFile(outdir + "/" + outfile);
    }

    public void dumpStateToFile(String outfile) {
      try {
        PrintWriter output =
          new PrintWriter(new FileWriter(new File(outfile)));
        int nModule = this.getModuleCount();
        for (int i = 0; i < nModule; i++) {
          output.printf("%f\n",this.getModule(i).getProbabilityBad());
        }
        output.close();
      }
      catch(IOException ioException) {
        System.out.println("Error writing file");
        System.exit(0);
      }
    }

    public String dumpTrueStateAsXML() {
      String result = "";
      result += "<true>";
      int nModule = this.getModuleCount();
      for (int i = 0; i < nModule; i++) {
        double state = 0.0;
        if ( this.getModule(i).hasDefect() ) state = 1.0;
        if ( i == nModule - 1 )
          result += String.format("%f", state);
        else
          result += String.format("%f,", state);
      }
      result += "</true>\n";
      return result;
    }

    public String dumpStateAsXML() {
      String result = "";
      result += String.format("<state>");
      int nModule = this.getModuleCount();
      int i;
      for (i = 0; i < nModule - 1; i++) {
        result += String.format("%f,",
          this.getModule(i).getProbabilityBad());
      }
      result += String.format("%f",
        this.getModule(i).getProbabilityBad());
      result += "</state>\n";
      return result;
    }

    //
    //  ... update methods ...
    //

    public void Reset() {
        int nModule = this.ModuleList.size();
        for (int i = 0; i < nModule; i++) {
            Module md = this.ModuleList.get(i);
            md.Reset();
       }
    }

    public void ResetBadValues() {
        int nModule = this.ModuleList.size();
        for (int i = 0; i < nModule; i++) {
            Module md = this.ModuleList.get(i);
            md.ResetBadValue();
       }
    }

    public void pushState() {
        int nModule = this.ModuleList.size();
        double[] state = new double[nModule];
        for (int i = 0; i < nModule; i++) {
          Module md = this.ModuleList.get(i);
          state[i] = md.getProbabilityBad();
       }
```

```
        this.StateStack.add(state);
    }

    public void popState() {
        int nModule = this.ModuleList.size();
        int nStack = this.StateStack.size();
        if ( nStack > 0 ) {
          double[] state =
            this.StateStack.remove(nStack - 1);
          for (int i = 0; i < nModule; i++) {
            Module md = this.ModuleList.get(i);
            double bi = state[i];
            md.setProbabilityBad(bi);
          }
        }
    }

    public void sortModulesByFailureRate() {
      Collections.sort(this.ModuleList, new byFailureRate());
    }



  class byFailureRate implements java.util.Comparator<Module> {

  public int compare(Module x, Module y) {

    int result = 0;

    if ( x.getFailureRate() > y.getFailureRate() )
      result = 1;
    else
      result = -1;

    return result;

  }

  } // end class byFailureRate


  public String describeTestsAsXML() {

       String result = "";

       for (int i=0; i < this.TestList.size(); i++) {
           Test t = this.TestList.get(i);
           result += String.format("<test>\n");
           result += String.format("  <name>%s</name>\n",t.getName());
           result += String.format("  <cost>%e</cost>\n",t.getCost());
           for (int j=0; j < t.ProbeList.size(); j++) {
              Probe p = t.ProbeList.get(j);
              Module m = p.getModule();
              result += String.format("  <probe>\n");
              result += String.format("    <device>%s</device>\n",
                                      m.getName());
              result += String.format("    <centerpoint>%f</centerpoint>\n",
                                      p.getCenterPoint());
              result += String.format("    <fraction>%f</fraction>\n",
                                      p.getFraction());
              result += String.format("  </probe>\n");
           }
           result += String.format("</test>\n");
       }

       return result;
    }

    public String describeModulesAsXML() {
        String result = "";

        for (int i=0; i < this.ModuleList.size(); i++) {
            Module m = this.ModuleList.get(i);
            result += String.format("<module>\n");
            result += String.format("  <name>%s</name>\n",m.getName());
            result += String.format("  <cost>%e</cost>\n",m.getCost());
            result += String.format("  <failurerate>%e</failurerate>\n",
                                  m.getFailureRate());
            result += String.format("</module>\n");
        }


        return result;

    }


    public String describeAsXML() {

        String result = "";

        result += "<systemobject>\n";

         result += this.describeModulesAsXML();

        result += this.describeTestsAsXML();

        result += String.format("</systemobject>\n");

        return result;

    }


    public void writeToXML(String outfile) {

       try {
         PrintWriter output =
          new PrintWriter(new FileWriter(new File(outfile),false));

         output.printf("<?xml version=\"1.0\"?>\n");
         output.printf("<systemobject>\n");

         for (int i=0; i < this.ModuleList.size(); i++) {
           Module m = this.ModuleList.get(i);
           output.printf("<module>\n");
           output.printf("  <name>%s</name>\n",m.getName());
           output.printf("  <cost>%e</cost>\n",m.getCost());
           output.printf("  <failurerate>%e</failurerate>\n",
             m.getFailureRate());
           output.printf("</module>\n");
         }
         for (int i=0; i < this.TestList.size(); i++) {
           Test t = this.TestList.get(i);
           output.printf("<test>\n");
          output.printf("  <name>%s</name>\n",t.getName());
          output.printf("  <cost>%e</cost>\n",t.getCost());
             for (int j=0; j < t.ProbeList.size(); j++) {
               Probe p = t.ProbeList.get(j);
               Module m = p.getModule();
               output.printf("  <probe>\n");
               output.printf("    <device>%s</device>\n",m.getName());
               output.printf("    <centerpoint>%f</centerpoint>\n",
                 p.getCenterPoint());
               output.printf("    <fraction>%f</fraction>\n",
                 p.getFraction());
               output.printf("  </probe>\n");
          }
        output.printf("</test>\n");
      }

      output.printf("</systemobject>\n");
      output.close();

    }
    catch(IOException ioException) {
      System.err.println("Error writing XML file");
      System.exit(0);
    }

  }

  public void readFromXML(String infile) {

      env.logWrite("... reading system object from %s",infile);

      this.removeAllProbes();
      this.ModuleList = new ArrayList<Module>();
      this.TestList   = new ArrayList<Test>();
      this.StateStack = new ArrayList<double[]>();

     Document dom = null;

      File xmlfile = new File(infile);

     DocumentBuilderFactory dbf =
         DocumentBuilderFactory.newInstance();

     try {
         DocumentBuilder db = dbf.newDocumentBuilder();
         dom = db.parse(xmlfile.toURI().toString());
     }
     catch (ParserConfigurationException pce) {
         pce.printStackTrace();
     }
     catch (SAXException se) {
         se.printStackTrace();
     }
     catch (IOException ioe) {
         ioe.printStackTrace();
     }

     Element docEle = dom.getDocumentElement();

     NodeList moduleList = docEle.getElementsByTagName("module");
     for (int i = 0; i < moduleList.getLength(); i++) {
         Element el = (Element) moduleList.item(i);
         String Name = (el.getElementsByTagName("name")).
            item(0).getFirstChild().getNodeValue();
         double FailureRate =
            Double.parseDouble((el.getElementsByTagName("failurerate")).
                              item(0).getFirstChild().getNodeValue());
         Module m = new Module(Name,FailureRate);
         this.addModule(m);
     }

     NodeList testList = docEle.getElementsByTagName("test");
     for (int i=0; i < testList.getLength(); i++) {
         Element el = (Element) testList.item(i);
         String Name = (el.getElementsByTagName("name")).
            item(0).getFirstChild().getNodeValue();
          Test t = new Test(Name);
         NodeList probeList = el.getElementsByTagName("probe");
         for (int j=0; j < probeList.getLength(); j++) {
            Element pl = (Element) probeList.item(j);
            String ModuleName = (pl.getElementsByTagName("device")).
```

```
                    item(0).getFirstChild().getNodeValue();
                double CenterPoint =
                    Double.parseDouble((pl.getElementsByTagName("centerpoint")).
                                      item(0).getFirstChild().getNodeValue());
                double Fraction =
                    Double.parseDouble((pl.getElementsByTagName("fraction")).
                                      item(0).getFirstChild().getNodeValue());
                 Module m = getModuleByName(ModuleName);
                 t.addProbe(m,CenterPoint,Fraction);
            }
             this.addTest(t);
        }

    } // readFromXML()


}
//
//  ======================
//  end class SystemObject
//  ======================
//
```

B.3 Module.java

```
//
//  ====================================================================
//
//  Module: Module.java
//
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2011
//  ====================================================================
//
package rbts.obj;

import java.util.ArrayList;
//
//  package: Module.java
//  --------------------
//
//    The Module represents the smallest replaceable unit within
//    the System.  We assume only a simple failure rate to describe
//    the reliability of the Module.
//
public class Module {
    //
    //  constant NO_DEFECT is used to indicate
    //  that our device is working correctly
    //
    public static final double NO_DEFECT = 9999.0;

    //
    //  ... attributes ...

    String Name;
    double FailureRate;
    double ProbabilityBad;
    double Defect;
    double Cost;
    int    FailureCount;
    ArrayList<Probe> ProbeList;

    //
    //  ... methods ...
    //

    //
    //  ... initialize a Module ...
    //
    public Module(String name, double frate) {
       //
       //  ... user supplied a Name and FailureRate
       //
        Name = name;
        FailureRate = frate;
       ProbabilityBad = frate;
       Cost = 1.0;
       Defect = NO_DEFECT;
        FailureCount = 0;
        ProbeList = new ArrayList<Probe>();
    }


    public String getName() {
      return this.Name;
    }

    public void setName(String name) {
       this.Name = name;
    }

    public void setCost(double cost) {
        this.Cost = cost;
    }

    public double getCost() {
        return this.Cost;
    }

    public double getFailureRate() {
       return this.FailureRate;
    }

    public void setFailureRate(double frate) {
       this.FailureRate = frate;
    }

    public double getProbabilityBad() {
      return this.ProbabilityBad;
    }

    public void setProbabilityBad(double bad) {
      if ( bad < 0.0 )
        this.ProbabilityBad = 0.0;
      else if ( bad > 1.0 )
        this.ProbabilityBad = 1.0;
      else
        this.ProbabilityBad = bad;
    }

    public void addProbe(Probe p) {
      this.ProbeList.add(p);
    }

    public Probe getProbe(int i) {
      return this.ProbeList.get(i);
    }

    public void removeProbe(int i) {
      this.ProbeList.remove(i);
    }

    public void removeAllProbes() {
      while ( this.ProbeList.size() > 0 )
        this.ProbeList.remove(0);
    }

    public int getProbeCount() {
      return this.ProbeList.size();
    }

    //
    //  ... plant a defect ...
    //
    public void setDefectAt(double p) {
       this.Defect = p;
        FailureCount++;
    }

    public int getFailureCount() {
      return this.FailureCount;
    }

    //
    //  ... remove the defect ...
    //
    public void Reset() {
       this.Defect = NO_DEFECT;
        this.ProbabilityBad = this.FailureRate;
    }

    public void ResetBadValue() {
        this.ProbabilityBad = this.FailureRate;
    }


    public boolean hasDefect() {
        if ( this.Defect == NO_DEFECT )
          return false;
        else
          return true;
    }


    public double getIntersectionBetween(Test t1, Test t2) {

      double result = 0.0;

      if ( (t1.getCoverageOn(this) == 0.0) ||
           (t2.getCoverageOn(this) == 0.0) )
        return 0.0;

      Probe pb1 = t1.getProbeOn(this);
      Probe pb2 = t2.getProbeOn(this);

      result = pb1.getIntersection(pb2);

      return result;

    }


    //
    //  ... test our Module within a specified arc
    //      about a specified center point
    //
    //      return TRUE if Module is defective
    //
    //      return FALSE if Module appears to be
    //        functioning normally
    //
    public boolean containsDefect(Coverage coverage) {
       //
       //  ... apply the test by "looking" within the
       //      user-specified arc for any defect
       //      returning TRUE if we are BAD and
       //      FALSE if we are GOOD or UNKNOWN
       //
        if ( coverage.containsPoint(this.Defect) ) {
           return true;
        }
        else
           return false;

    } // end containsDefect()


    //
    //  computeBadGivenFail
    //  -------------------
    //
    //    Compute the probability that given test Tj = FAIL
    //    our module is BAD
    //
    //    P(Bi|Fj) =               P(Fj|Bi)P(Bi)
    //                      ------------------------------
    //                       P(Fj|Bi)P(Bi) + P(Fj|Gi)P(Gi)
    //
    //
    //    where:
    //
    //      P(Bi) = current probability Mi is BAD
    //
    //      P(Gi) = 1 - P(Bi)
    //
    //      P(Fj|Bi) = probability of detecting failure, or
    //                 the coverage alpha_ij
    //
    //      P(Fj|Gi) = probability that even though Mi is good
    //                 some other module failed and was detected
    //                 by Tj
    //
    public double computeBadGivenFail(Test t) {
        double alpha = t.getCoverageOn(this);
        if ( alpha == 0.0 ) return this.ProbabilityBad;
        double bi = this.ProbabilityBad;
        double gi = 1.0 - bi;
        double fjbi = alpha;
         int nProbe = t.getProbeCount();
        double prod = 1.0;
        for (int i=0; i < nProbe; i++) {
            Probe pb = t.getProbe(i);
            Module md = pb.getModule();
            if ( md != this )
               prod = prod*(1.0 - t.getCoverageOn(md)*md.getProbabilityBad());
        }
        double fjgi = 1 - prod;
        double result = (fjbi*bi) / (fjbi*bi + fjgi*gi);
        return result;
    }

    //
    //  computeBadGivenPass
    //  -------------------
    //
    //    Compute the probability that given test Tj = PASS
    //    our module is BAD
    //
    //    P(Bi|Pj) =               P(Pj|Bi)P(Bi)
    //                      ------------------------------
    //                       P(Pj|Bi)P(Bi) + P(Pj|Gi)P(Gi)
    //
    //
    //    where:
    //
    //      P(Bi) = current probability Mi is BAD
    //
    //      P(Gi) = 1 - P(Bi)
    //
    //      P(Pj|Bi) = probability of non-detect of failure
    //                 or 1 - alpha_ij
    //
    //      P(Pj|Gi) = probability some other module failed
    //                 in the set of modules covered by Tj
    //
    public double computeBadGivenPass(Test t) {
        double alpha = t.getCoverageOn(this);
         if ( alpha == 0.0 ) return this.ProbabilityBad;
        double bi = this.ProbabilityBad;
        double gi = 1.0 - bi;
        double pjbi = 1.0 - alpha;
         int nProbe = t.getProbeCount();
        double prod = 1.0;
        for (int i=0; i < nProbe; i++) {
            Probe pb = t.getProbe(i);
            Module md = pb.getModule();
            if ( md != this )
                prod = prod*(1.0 - t.getCoverageOn(md)*md.getProbabilityBad());
        }
        double pjgi = prod;
        double result = (pjbi*bi) / (pjbi*bi + pjgi*gi);
        return result;
    }

}
//
//  ===================
//  end package: Module
//  ===================
//
```

THIS PAGE INTENTIONALLY LEFT BLANK

B.4 Test.java

```
//
//  ====================================================================
//
//  Module: Test.java
//
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2011
//  ====================================================================
//
package rbts.obj;

import java.util.ArrayList;
//
//  package: Test.java
//  ------------------------
//
//  A Test is the smallest diagnostic executable within the System.
//
//  We treat a Test as a collection of Probes on Modules, with each
//  Probe testing some fraction on the interval [0,1) over a
//  specific Module.
//
//  When a Test is executed, only one result is returned.
//  The convention is
//
//    PASS: No faulty devices indicated
//
//    FAIL: At least one faulty device indicated
//
//  This result is the aggregation of all Probes applied
//  so that, for example, if a Test covers five Modules and the
//  Test returns TRUE, we know at least one of those five devices
//  is faulty.
//
public class Test {

    public enum Result { PASS, FAIL }

    //
    //  ... attributes ...
    //
    String Name;
    double Cost;
    ArrayList<Probe> ProbeList;

    //
    //  ... methods ...
    //
    public Test(String name) {
       this.ProbeList = new ArrayList<Probe>();
       this.setName(name);
        this.setCost(1.0);
    }

    public void setName(String name) {
       this.Name = name;
    }

    public String getName() {
       return this.Name;
    }
    public void setCost(double c) {
        this.Cost = c;
    }

    public void setCostByCoverage() {
        double result = 0.0;
        int ncount = this.getProbeCount();
        for (int i=0; i < ncount; i++) {
          Probe pb = this.ProbeList.get(i);
          result += pb.getFraction();
        }
        this.setCost(result);
    }

    public double getCost() {
        return this.Cost;
    }

    public void addProbe(Module m, double cp, double cov) {
       Probe pb = new Probe(this,m,cp,cov);
        boolean Inserted = false;
        for (int i = 0; i < this.ProbeList.size(); i++) {
          Probe pbi = this.ProbeList.get(i);
          if (m == pbi.getModule()) {
            // System.out.printf("debug: replacing probe on %s\n",
            //    m.getName());
            this.ProbeList.set(i,pb);
            Inserted = true;
          }
        }
        if ( !Inserted ) {
          this.ProbeList.add(pb);
          m.addProbe(pb);
        }
    }

    public Probe getProbe(int i) {
       return this.ProbeList.get(i);
    }


    public Probe getProbeOn(Module md) {
      Probe pb = null;
      int nProbe = this.ProbeList.size();
      for (int i=0; i < nProbe; i++) {
        Probe pbx = this.ProbeList.get(i);
        if ( pbx.getModule() == md )
          pb = pbx;
      }
      return pb;
    }


    public void removeProbe(int i) {
       this.ProbeList.remove(i);
    }

    public void removeAllProbes() {
         while ( this.ProbeList.size() > 0 )
            this.ProbeList.remove(0);
    }

    public int getProbeCount() {
      return this.ProbeList.size();
    }

    public Test.Result applyTest() {
       Test.Result result = Test.Result.PASS;
```

```
        int i;
        int n = this.ProbeList.size();
        for (i = 0; i < n; i++) {
            Probe pb = this.ProbeList.get(i);
            Probe.Result pbres = pb.applyProbe();
            if ( pbres == Probe.Result.FAIL )
               result = Test.Result.FAIL;
        }
        return result;
    }


    public ArrayList<Module> getModulesProbed() {
        ArrayList<Module> dlist = new ArrayList<Module>();
        int nProbe = this.ProbeList.size();
        int i;
        for (i = 0; i < nProbe; i++) {
            Probe p = this.ProbeList.get(i);
            Module d = p.getModule();
            dlist.add(d);
        }
        return dlist;
    }

    public double getCoverageOn(Module m) {
        int nProbe = this.ProbeList.size();
        double result = 0.0;
        for (int i=0; i < nProbe; i++) {
            Probe pb = this.ProbeList.get(i);
            if ( m == pb.getModule() )
               result = pb.getFraction();
        }
         if ( result > 1.0 ) {
           System.out.printf("WARNING: coverage on module %s = %f\n",
             m.getName(), result);
         }
        return result;
    }

    //
    //  probabilityPass()
    //  -----------------
    //
    //    Return the probability that a test will pass
    //    based on the coverage of the test and the
    //    a priori probability the covered modules are bad
    //
    public double probabilityPass() {
        double result = 1.0;
        ArrayList<Probe> pblist = this.ProbeList;
        for (int i=0; i < pblist.size(); i++) {
            Probe pb = pblist.get(i);
            double b = pb.getModule().getProbabilityBad();
            double a = pb.getFraction();
            result = result * ( 1.0 - a*b );
        }
        return result;
    }

    //
    //  probabilityFail()
    //  -----------------
    //
    //   Return probability that a test will fail
    //
    public double probabilityFail() {
      return ( 1.0 - this.probabilityPass() );
    }
```

```
}
//
//  ===================
//   end package: Test
//  ===================
//
```

B.5 Coverage.java

```
//
//  ======================================================================
//  Project: Risk-based Testing Simulation
//           Pfeiffer, Kanevsky, Housel
//           Department of Information Sciences
//           Naval Postgraduate School
//
//  Date:    1 Oct 2011
//  ======================================================================
//
package rbts.obj;

import java.util.ArrayList;
//
//  package: Coverage.java
//  ----------------------
//
//    The Coverage class is used to model the relationship between
//    the System objects Test and Module.
//
//    A Coverage object represents the arc a_{ij} on module M_i
//    which is inspected by test T_j.  If T_j has no coverge on
//    M_i this arc is null with measure zero.
//
public class Coverage {
    //
    //  -----------------------
    //  private class: Interval
    //  -----------------------
    //
    private class Interval {

        double Left;
        double Right;

        Interval(double left, double right) {
            this.Left = left;
            this.Right = right;
        }

         void set(double left, double right) {
            this.Left = left;
            this.Right = right;
         }

        double measure() {
            double result = (this.Right - this.Left);
            return result;
        }

        boolean containsPoint(double p) {
            if ( this.Left <= p && p <= this.Right )
               return true;
            else
               return false;
        }

    }
    //  -------------------
    //  end class: Interval
    //  -------------------
    //
    //  ---------------------------
    //  private class: IntervalList
    //  ---------------------------
    //
```

```
     private class IntervalList {

        ArrayList<Interval> List;

        IntervalList() {
            List = new ArrayList<Interval>();
        }

         void compress() {

            int nList = this.List.size();

            boolean SORTED = false;
            while ( !SORTED ) {
                SORTED = true;
               nList = this.List.size();
               for (int i=0; i < nList - 1; i++) {
                   Interval v  = this.List.get(i);
                   Interval vp = this.List.get(i+1);
                    if ( v.Left > vp.Left ) {
                      this.List.set(i,vp);
                      this.List.set(i+1,v);
                      SORTED = false;
                    }
                }
            }

            boolean DONE = false;
            while ( !DONE ) {

               DONE = true;
               nList = this.List.size();
               for (int i=0; i < nList - 1; i++) {
                   Interval v  = this.List.get(i);
                   Interval vp = this.List.get(i+1);
                   //
                   //  v contains v' (skip)
                   //
                   if ( v.Left <= vp.Left && v.Right >= vp.Right ) {
                       this.List.remove(i+1);
                       nList--;
                       DONE = false;
                   }
                   //
                   //  v' contains v (adjust v)
                   //
                   else if ( v.Left >= vp.Left  && v.Right <= vp.Right ) {
                       this.List.remove(i);
                       nList--;
                       DONE = false;
                   }
                   //
                   //  v' overlaps v on the left (adjust v)
                   //
                   else if ( vp.Left  <= v.Left &&
                         vp.Right >= v.Left &&
                         vp.Right <= v.Right ) {
                       v.Left = vp.Left;
                       this.List.remove(i+1);
                       nList--;
                       DONE = false;
                   }
                   //
                   //  v' overlaps v on the right (adjust v)
                   //
                   else if ( vp.Left  >= v.Left &&
                         vp.Left  <= v.Right &&
                         vp.Right >= v.Right) {
                       v.Right = vp.Right;
```

```
                  this.List.remove(i+1);
                  nList--;
                  DONE = false;
               }

           } // end for(i)

        } // end while(!DONE)

    } // end compress()


    void addInterval(double left, double right) {

         List.add(new Interval(left, right));

         this.compress();

    } // end addInterval()


     void addInterval(Interval v) {
       double left = v.Left;
       double right = v.Right;
       this.addInterval(left,right);
     }

    void addArc(double center, double fraction) {

        double left = center - fraction/2.0;
        double right = center + fraction/2.0;

         if ( left < 0.0 ) {
           this.addInterval(0.0, right);
           this.addInterval(1.0 + left, 1.0);
         }
         else if ( right > 1.0 ) {
           this.addInterval(0.0, right - 1.0);
           this.addInterval(left, 1.0);
         }
         else
           this.addInterval(left, right);

    } // end addArc()

    Interval get(int k) {
        return List.get(k);
    }

    boolean containsPoint(double point) {

        int nSize = List.size();

        for (int i = 0; i < nSize; i++)
           if ( this.List.get(i).containsPoint(point) ) return true;

        return false;
    }

    double measure() {
        double sum = 0.0;
        int nSize = List.size();
        for (int i = 0; i < nSize; i++)
           sum += List.get(i).measure();
        return sum;
    }

     int size() {
         return this.List.size();
        }

       String getCoverageGraphic(int nGraphic) {

           char[] coverage = new char[nGraphic];
           for (int i=0; i < nGraphic; i++) coverage[i] = '.';

           for (int k=0; k < this.List.size(); k++) {
              Interval v = this.get(k);
              int left  = (int) (nGraphic * v.Left + 0.5);
              int right = (int) (nGraphic * v.Right + 0.5);
              for (int p=left; p < right; p++)  coverage[p] = 'x';
           }

           String coverageGraphic = new String(coverage);

           return coverageGraphic;

       }
    }
    //
    //  ------------------------------
    //  end private class IntervalList
    //  ------------------------------
    //
    //  ---------------------------
    //  main body of class Coverage
    //  ---------------------------
    //
    IntervalList List;

    public Coverage (double centerpoint, double fraction) {
       List = new IntervalList();
       this.List.addArc(centerpoint,fraction);
    }

    double measure() {
       return this.List.measure();
    }

    double intersection(Coverage that) {

      IntervalList diff = new IntervalList();

      double a = this.measure();

      double b = that.measure();

      for (int i=0; i < this.List.size(); i++) {
        Interval v = this.List.get(i);
        diff.addInterval(v);
      }

      for (int i=0; i < that.List.size(); i++) {
        Interval v = that.List.get(i);
        diff.addInterval(v);
      }

      double result = (a + b) - diff.measure();

      return result;

    }

    boolean containsPoint(double p) {
       return this.List.containsPoint(p);
    }

    String getCoverageGraphic(int nGraphic) {
        return this.List.getCoverageGraphic(nGraphic);
    }
    //
    //  end class: Coverage
    //

}
//
//  =====================
//  end package: Coverage
//  =====================
//
```

THIS PAGE INTENTIONALLY LEFT BLANK

B.6 Probe.java

```
//
// ======================================================================
// Project: Risk-based Testing Simulation
//          Pfeiffer, Kanevsky, Housel
//          Department of Information Sciences
//          Naval Postgraduate School
//
// Date:    1 Oct 2011
// ======================================================================
//
// package Probe.java
// ------------------
//
//   Each Probe describes the coverage of a Test on a specific
//   Module, using a center point and fraction (or arc length)
//   to describe the portion of the Module exercised when a
//   specific Test is applied.
//
//   A Test is a collection (ArrayList) of Probes.
//
package rbts.obj;

public class Probe {

    public enum Result { PASS, FAIL }

    //
    //  ... attributes ...
    //
    Module module;
    Test test;
    double CenterPoint;
    double Fraction;
    Coverage coverage;

    //
    //  ... methods ...
    //
    public Probe (Test t, Module m, double cp, double f) {
        test = t;
       module = m;
        Fraction = f;
        CenterPoint = cp;
        coverage = new Coverage(cp, f);
    }

    public Probe.Result applyProbe() {
       if ( this.module.containsDefect(this.coverage) )
           return Probe.Result.FAIL;
       else
           return Probe.Result.PASS;
    }

    public Module getModule() {
       return this.module;
    }

    public Test getTest() {
      return this.test;
    }

    public double getFraction() {
        return this.coverage.measure();
    }

    public double getCenterPoint() {
        return this.CenterPoint;
    }

    public Coverage getCoverage() {
        return this.coverage;
    }

    public double getIntersection(Probe that) {
      Module mdthis = this.getModule();
      Module mdthat = that.getModule();
      if ( mdthis != mdthat ) return 0.0;
      Coverage cvthis = this.getCoverage();
      Coverage cvthat = that.getCoverage();
      double result = cvthis.intersection(cvthat);
      return result;
    }

    public String getCoverageGraphic(int nGraphic) {
        String result = coverage.getCoverageGraphic(nGraphic);
       return result;
    }

}
//
//  ==================
//  end package: Probe
//  ==================
//
```

2003 - 2011 Sponsored Research Topics

Acquisition Management

- Acquiring Combat Capability via Public-Private Partnerships (PPPs)
- BCA: Contractor vs. Organic Growth
- Defense Industry Consolidation
- EU-US Defense Industrial Relationships
- Knowledge Value Added (KVA) + Real Options (RO) Applied to Shipyard Planning Processes
- Managing the Services Supply Chain
- MOSA Contracting Implications
- Portfolio Optimization via KVA + RO
- Private Military Sector
- Software Requirements for OA
- Spiral Development
- Strategy for Defense Acquisition Research
- The Software, Hardware Asset Reuse Enterprise (SHARE) repository

Contract Management

- Commodity Sourcing Strategies
- Contracting Government Procurement Functions
- Contractors in 21st-century Combat Zone
- Joint Contingency Contracting
- Model for Optimizing Contingency Contracting, Planning and Execution
- Navy Contract Writing Guide
- Past Performance in Source Selection
- Strategic Contingency Contracting
- Transforming DoD Contract Closeout
- USAF Energy Savings Performance Contracts
- USAF IT Commodity Council
- USMC Contingency Contracting

Financial Management

- Acquisitions via Leasing: MPS case
- Budget Scoring
- Budgeting for Capabilities-based Planning
- Capital Budgeting for the DoD
- Energy Saving Contracts/DoD Mobile Assets
- Financing DoD Budget via PPPs
- Lessons from Private Sector Capital Budgeting for DoD Acquisition Budgeting Reform
- PPPs and Government Financing
- ROI of Information Warfare Systems
- Special Termination Liability in MDAPs
- Strategic Sourcing
- Transaction Cost Economics (TCE) to Improve Cost Estimates

Human Resources

- Indefinite Reenlistment
- Individual Augmentation
- Learning Management Systems
- Moral Conduct Waivers and First-term Attrition
- Retention
- The Navy’s Selective Reenlistment Bonus (SRB) Management System
- Tuition Assistance

Logistics Management

- Analysis of LAV Depot Maintenance
- Army LOG MOD
- ASDS Product Support Analysis
- Cold-chain Logistics
- Contractors Supporting Military Operations
- Diffusion/Variability on Vendor Performance Evaluation
- Evolutionary Acquisition

- Lean Six Sigma to Reduce Costs and Improve Readiness
- Naval Aviation Maintenance and Process Improvement (2)
- Optimizing CIWS Lifecycle Support (LCS)
- Outsourcing the Pearl Harbor MK-48 Intermediate Maintenance Activity
- Pallet Management System
- PBL (4)
- Privatization-NOSL/NAWCI
- RFID (6)
- Risk Analysis for Performance-based Logistics
- R-TOC AEGIS Microwave Power Tubes
- Sense-and-Respond Logistics Network
- Strategic Sourcing

Program Management

- Building Collaborative Capacity
- Business Process Reengineering (BPR) for LCS Mission Module Acquisition
- Collaborative IT Tools Leveraging Competence
- Contractor vs. Organic Support
- Knowledge, Responsibilities and Decision Rights in MDAPs
- KVA Applied to AEGIS and SSDS
- Managing the Service Supply Chain
- Measuring Uncertainty in Earned Value
- Organizational Modeling and Simulation
- Public-Private Partnership
- Terminating Your Own Program
- Utilizing Collaborative and Three-dimensional Imaging Technology

A complete listing and electronic copies of published research are available on our website: www.acquisitionresearch.org

THIS PAGE INTENTIONALLY LEFT BLANK